Barbara Gigerl

Efficient and Secure Masking Schemes to Counteract Power Analysis Attacks in Practice

DOCTORAL THESIS
to achieve the university degree of
Doctor of Technical Sciences; Dr. techn.

submitted to
Graz University of Technology

Assessors
Prof. Stefan Mangard
Graz University of Technology, Austria

Prof. Tim Güneysu
Ruhr-University Bochum, Germany

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Graz, June 2024

Abstract

Embedded and IoT devices rely on cryptographic building blocks to protect sensitive user data from unrestricted access. Cryptographic algorithms have been designed to provide security against mathematical attacks, assuming that the adversary knows the input, output, and the algorithm itself but not the secret key. Real-world attackers are often more powerful because they can exploit physical access to the device, allowing them to observe physical properties like the device's power consumption. Using side-channel attacks, such as differential power analysis, these physical properties can be statistically analyzed to extract the secret key. One popular countermeasure is masking, which splits sensitive values into multiple random shares, effectively decoupling the sensitive data from the data processed by the device and, thus, the power consumption. The security of masking schemes is based on the independent leakage assumption (ILA), which states that independent computations result in independent leakage. Unfortunately, it has been shown that the ILA does not always hold for masked implementations that are used in practice. In this thesis, we work on improving the security and efficiency of masked implementations in software and hardware.

First, we study the security of masked software when executed by a microprocessor and identify several microarchitectural building blocks that could prevent leakage-free execution due to ILA violations. To fix the discovered issues, we explore possible solutions on the hardware and software level and compare them with respect to efficiency. We focus on simple as well as more complex CPUs, which include multiple pipeline stages and superscalar building blocks. Furthermore, we investigate the security of masked software when running as a task in an operating system. To identify leakage in the first place, we propose a new formal verification approach that allows to verify the execution of masked software implementations directly on the CPU netlist, facilitating the detection of ILA violations stemming from the CPU microarchitecture.

Second, we focus on masked hardware implementations. Typically, these implementations compensate for ILA violations either by consuming a significant amount of fresh randomness or by an increased encryption latency. We research strategies to lower the randomness consumption of masking in hardware without increasing the latency, mainly by reusing fresh randomness for unrelated computations during the encryption. In addition, we research new formal verification concepts for masked hardware implementations and apply these techniques in different contexts. We construct a verification approach that supports both Boolean and arithmetic masking and allows us to detect ILA violations in implementations adapting both types of masking. In the context of masked hardware implementations on FPGAs, we build a new verification tool that can be used to uncover leakage introduced by the FPGA synthesis process.

Long live all the mountains we moved
I had the time of my life fighting dragons with you

Taylor Swift – Long Live (Taylor's Version)

Acknowledgements

Many people contributed to this dissertation in many different ways. In the following, I want to take the moment and express my deepest gratitude to all of you.

First, I want to thank my advisor Stefan Mangard for giving me the opportunity to start a PhD. It was a pleasure to work with you and I really enjoyed all the interesting and inspiring discussions we had. Thank you for making sesys such a great place to work and grow and for all your guidance in the scientific world that led me toward researching a topic I truly enjoy. Last but not least, thank you for providing constructive and helpful feedback on a draft of this thesis that really helped to improve it.

I want to thank Tim Güneysu for agreeing to be part of my doctoral examination committee and for his valuable comments on this thesis.

Thanks to all my amazing co-authors for the successful joint work and all the exciting projects we've been doing together. I want to thank all the current and past members of the sesys group and my colleagues from IAIK - I was enchanted to meet you! I owe special thanks to Robert Primas. Thank you for your support and guidance, especially when I started my PhD, and for all the time you invested in our joint research projects. It was a great time sharing an office with you, and I will always be grateful for you providing new perspectives on problems I believed to be unsolvable.

I am also eternally thankful for my dear family: my parents, who always believed in whatever I did, my brothers and their families, and my grandmother. I also want to thank all my friends, including Tanja (for understanding) and David (for giving me the best advice ever: "Don't cry, write!")

The challenges of this dissertation clearly extend beyond a purely academic scope, as it also has represented a significant personal hurdle for me. I want to thank the team of the PSB Graz for the helpful discussions and suggestions.

Finally, I want to thank my amazing boyfriend Alex for a list of things that is too long to write down. I am truly the lucky one, having you in my life, and I could not imagine having done this thesis without you - it was legendary! Thank you for tossing me the car keys, for taking me to Florida, for working the graveyard shift, for burnt toast (Sundays), for driving the getaway car, for painting the kitchen neon, and for taking on the world together. I hope that I can go where you go and that we can always be this close (forever and ever).

Contents

Part I.

Efficient and Secure Masking Schemes to Counteract Power Analysis Attacks in Practice

The greatest of luxuries is your secrets.
Taylor Swift – Dear Reader

1 Introduction

The Internet of Things (IoT) is becoming increasingly intertwined with our daily lives, offering comfortable and easy solutions to our everyday problems. While in 2003, approximately 500 million devices were connected to the internet, already in 2010, the number of connected devices (12.5 billion) by far outgrew the number of humans living on the planet (6.8 billion) [Eva11]. Reasons for this steep increase are the introduction of ubiquitous devices like smartphones and tablets and the growing ability of everyday objects to communicate with the world around them. In 2024, it is almost impossible to imagine the world without electronic door locks, cameras automatically recognizing the number plate of our cars when we pass the tollbooth and the possibility of tracking goods in real-time in supply chain management. However, the sheer convenience and benefits of using IoT technology often make us blind to the possible risks to our privacy. It is in the nature of IoT devices to collect, process, and transmit our data, which is in most cases sensitive corporate or personal information, for example, our heart rate monitored by fitness trackers or information about the electricity consumption of our home sent by smart meters to the grid operators. Since leaking this data to the outside world can have serious monetary, legal, or safety consequences, a key aspect of implementing IoT devices is protecting them from unauthorized access.

Cryptography One of the most critical measures to secure communication between two parties, e.g., an IoT device and a server, is cryptography. One general tool provided by cryptography are encryption algorithms (ciphers) that allow to transform a message (plaintext) into a concealed message (ciphertext) using a key. In the case of symmetric cryptography, the communicating parties share a common secret key k, which is only known to them and is used for both encryption and decryption. To securely transmit a message, the sender encrypts the message m with the algorithm E using the key k and obtains the ciphertext

$c = E_k(m)$. The sender then sends the ciphertext c to the receiver, who uses the same key to decrypt the message and obtains the plaintext $m = E_k^{-1}(c)$. The encryption algorithm is designed in a way such that any eavesdropper who observes c cannot practically obtain any information about k or m, which refers to security in the so-called *black-box* model. In the black-box model, the adversary can control the input plaintext and observe the plaintext-ciphertext pairs for a certain amount of encryptions. Additionally, the adversary knows which cryptographic algorithm is used but has no knowledge about the internal cipher state. Over the years, security in the black-box model has been the focus when designing cryptographic primitives, which covers mathematical and statistical attacks like differential or linear cryptanalysis [BS90; BS91; Mat93].

Physical Attacks In the context of the IoT, symmetric cryptography is used by cryptographic devices such as smart cards, which store a cryptographic key and perform cryptographic operations using these keys [MOP07]. The cryptographic key needs to be kept secret. If the key falls into the hands of an attacker, the possible consequences are devastating, including data breaches, identity theft, and financial loss. For example, the cryptographic key stored on a prepaid credit card could be misused to increase the balance or issue malicious money transactions.

Compared to the traditional setting, an attacker in the IoT world often has more powerful abilities, effectively turning the black box into a gray box. While the black-box model assumes that the only insecure component in the cryptographic system is the transmission channel and that the operations performed by the cryptographic devices are entirely hidden, this is not necessarily the case in reality. In many scenarios, attackers might be able to gain physical access and perform physical rather than mathematical attacks. Especially IoT devices make perfect targets for physical attacks due to their ubiquitous nature. For example, attackers can quickly gain access to electronic key cards, e.g., by just stealing them. For big server machines located in a company's server room, this is much harder. In general, physical access means that the attacker possesses the cryptographic device or is in close vicinity and can observe and record its physical properties, which are referred to as side-channel information. The idea of exploiting side-channel information is that cryptographic devices perform computations involving the secret key, and computations, in turn, strongly influence physical properties. Consequently, a dependency exists between the secret key and the side-channel information that can be analyzed to recover the key. From an attacker's perspective, the cryptographic primitive is then found in a *gray-box* setting because some information about the internal cipher state is leaked through side-channel information.

Side-channel attacks using various kinds of physical properties have been used to extract secrets from cryptographic devices, including time [Koc96; OST06; SWT01], electromagnetic radiation [GMO01; Hey+12; QS01], sound [Gen+19; GST14], temperature [HS13], and photonic emission [FH08; Sch+12; Sch+13;

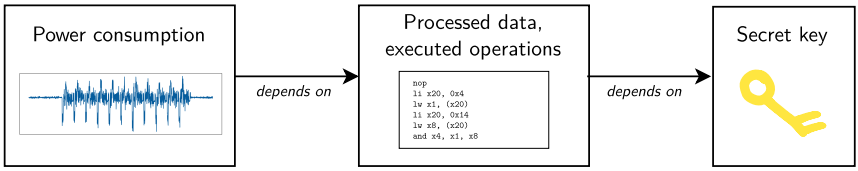


Figure 1.: Dependency chain between secret key and power consumption of a cryptographic device (extended from [MOP07])

Sko09]. In 1999, Kocher et al. [KJJ99] proposed power analysis attacks that utilize the instantaneous power consumption of CMOS circuits as side-channel information. CMOS is the prevailing technology for building ICs like ASICs, FPGAs, and microprocessors because of its low overall power consumption. Power analysis attacks exploit a specific characteristic of CMOS circuits: instantaneous power consumption is primarily determined by switching activity, such as a register or wire changing its logical value. As a result, different operations on different data cause different switching activity and, thus, different power consumption. Differential Power Analysis (DPA) is one of the most popular methods to exploit power consumption as a side-channel. To perform DPA, the attacker statistically analyzes the differences in the power consumption over multiple executions of the encryption to obtain information about intermediate values processed by the cryptographic device, which depend on the secret key.

Countermeasures In order to counteract power analysis attacks like DPA, it is necessary to identify the requirements for a successful attack. One fundamental requirement is the dependency between the power consumption and the secret key, as shown in Figure 1. During the execution of a cryptographic algorithm, the secret key is leaked by the processed data and the executed operations via the power consumption. To prevent the leakage of the key, either the link between the power consumption and the data/operations or the link between the data/operations and the secret key must be broken. There exist two general classes of countermeasures, hiding and masking, which focus on breaking either one of these two links.

Hiding aims at weakening the dependency between the power consumption and the processed data and executed operations by randomizing or equalizing the power consumption. Randomization of the power consumption can be achieved by shuffling the execution order of operations, inserting random delays or dummy operations, or utilizing noise engines to produce random switching activity. Another possibility is equalizing the power consumption such that every operation for every data value consumes approximately the same amount of power, e.g., by using dual-rail logic styles. Several proposals to protect cryptographic devices by hiding exist [CCD00; Das+17; GM11b; HOM06; Man04; MSS09;

RPD09; TV04].

The goal of masking [Cha+99; GP99] is to remove the dependency between the secret key and the processed data and operations. This is achieved by randomizing sensitive intermediate values, such as the key, by splitting them randomly into multiple *shares*. The cryptographic algorithm itself is adapted to process each share individually instead of the secret key. Although the power consumption of the cryptographic device still depends on the processed data, the processed data is effectively random. Every share is statistically independent of the sensitive value, such that the sensitive value can only be recovered by recovering all shares, which makes attacks much more difficult and often impractical. The security of a masked cryptographic algorithm is determined by the security order d, which is also referred to as the masking order. For instance, first-order masking ($d = 1$) works with at least two shares and protects against first-order DPA attacks, while protection against dth-order attacks requires dth-order masking, and thus, at least $d + 1$ shares. In the context of symmetric cryptography, Boolean masking is commonly used, which uses the exclusive or ($\oplus$) operator to split a value into multiple shares. Other areas, for instance, some PQC schemes, require a mix of arithmetic masking based on modular additions and Boolean masking. Compared to hiding, masking does not require modifications to the power consumption characteristics of the physical device since it is applied exclusively on algorithmic level. In this thesis, we focus on the masking countermeasure.

Besides hiding and masking, countermeasures can be implemented on protocol level, such as re-keying, which ensures that a single key is only used for one or very few encryptions. Re-keying is a central building block for the design of leakage-resilient modes of cryptographic primitives [DP08], which aim at limiting the amount of side-channel information an attacker is able to collect. However, in many scenarios, frequent updates of the secret key are not possible. For instance, smart cards are usually assigned their secret key upon production, and no mechanism exists to update it.

The Independent Leakage Assumption The security of a masking scheme against DPA attacks can be formally proven. This is achieved by verifying that the distribution of every processed intermediate does not depend on the sensitive value. In the case of a masking scheme working with $d + 1$ shares, every tuple of d or less intermediate variables must be independent of the sensitive variable to be secure. For example, to split a sensitive value s into two shares s_1 and s_2, one would sample s_1 randomly from a uniform distribution and compute $s_2 = s \oplus s_1$. Clearly, s_1 is independent of s, but also s_2 is independent because s is concealed (*masked*) by s_1. Many works give formal proofs of the security of masking schemes [Bar+18; Cor+13; Gou01; NSS22; PR13], including, for instance, the work of Rivain et al. [RP10], who propose a generic dth-order masking of the AES along with a manually derived formal security proof.

Formal security proofs have in common that they are based on the *independent*

leakage assumption (ILA) [Ren+11]. The ILA states that independent computations lead to independent leakage, i.e., the leakage caused by two distinct computations depends on at most one intermediate value but not on their combinations. For example, on a microprocessor, the ILA implies that the power consumption of two instructions executed successively is independent of each other. To obtain a secure masking scheme that meets the properties derived in the formal proof, it must be ensured that the ILA holds.

Masking in Practice Based on the algorithmic description of a masking scheme of a cryptographic primitive, the masking scheme is either implemented in hardware or in software. For instance, to create a hardware implementation, masking schemes are often coded in a Hardware Description Language (HDL) like Verilog and then manufactured into an ASIC or ported to an FPGA. Masked software implementations are frequently created using a programming language like C or assembly and then executed by a CPU, either in bare-metal mode or within an operating system. ASICs, FPGAs, and CPUs are typically built using CMOS technology, which is prone to physical effects like glitches, transitions, and couplings. In a CMOS circuit, at the beginning of a clock cycle, the data stored by registers is propagated into the combinatorial logic of the circuit. Due to differences in the wire lengths and propagation delays of gates, some gates might temporarily compute incorrect results when one gate input has arrived but the other has not. These temporary events are known as glitches and transitions. Coupling effects include crosstalk between adjacent wires, power supply noise, or IR drop.

Numerous publications demonstrate that the ILA is violated by physical effects in CMOS circuits [Cnu+17; Dho21; FG05; GMK16; ISW03; LBS19; MPG05; MPO05; Rep+15], leading to insecure masked hardware and software implementations. For example, combining two shares s_1 and s_2 with a random value r by the expression $x = (s_1 \oplus r) \oplus s_2$ is valid in theory. In a masked hardware implementation representing the expression as a CMOS circuit, r could be delayed, and x would temporarily compute $s_1 \oplus s_2$, leading to the combined instead of independent leakage of the two shares. In the case of masked software implementations, physical effects occurring in the CPU on the hardware level can lead to violations of the ILA [Bal+14; Cor+12; Gro+16a; MMT20; PV17]. For instance, overwriting a CPU register that stores one share with its counterpart can leak the Hamming distance between the two shares.

Verification of Masked Implementations To detect potential violations of the ILA, after creating a masked implementation from the theoretical description of a masking scheme, designers need to test if the security order in practice adheres to the theoretical protection order. One option is to perform empirical verification using leakage assessments such as Welch's t-test or concrete attacks like DPA. For masked hardware implementations, this can be done by producing an ASIC

test chip or evaluating the design on an FPGA board. For masked software, it is possible to assess the implementation directly on the microcontroller or use a leakage simulator that tries to estimate the power leakage for a specific microprocessor [Har+03; MOW17; She+21a; She+21b].

Formal verification approaches analyze the masked implementation for ILA invalidations and generate a security proof in case the implementation is secure. This analysis can be performed in an automated way by a formal verification tool with respect to a specific attacker model. For example, existing approaches for masked hardware circuits often work by investigating the implementation in the robust probing model [Fau+18; ISW03]. In the robust probing model, the attacker possesses d probes, which allow the observation of values of up to d wires in a masked circuit. The probes are parametrized by a tuple (g, t, c), which defines whether glitches ($g = 1$), transitions ($t = 1$), or coupling ($c = 1$) effects can be observed by a probe. A masked hardware circuit is dth-order secure in the robust probing model if an attacker, who places d probes on arbitrary wires in the circuit, cannot infer any information about the secret value by combining these observations.

1.1. Objectives and Contributions

The objective of this cumulative thesis is to improve the security and efficiency of masked implementations in software and hardware. In Section 1.1.1 and Section 1.1.2, we describe the main contributions made in each area. Part II contains the scientific publications that are part of the author's thesis. Each work is preceded by the publication details and a brief description of the author's contribution. The content of the papers is unmodified from the camera-ready versions published at the respective conferences, but the format was modified to fit the layout of this thesis.

1.1.1. Masking in Software

Masked software implementations that have been proven secure in theory often exhibit leakage in practice when being executed by a CPU. This is due to physical effects, like glitches and transitions, that occur in the CPU on hardware level and violate the ILA [Bec+22; Cor+12; Gro+16a; MMT20; PV17]. Based on this, Balasch et al. [Bal+14] formulate the so-called order reduction theorem, which states that a masking scheme that has been proven to be dth-order secure in theory is only $\lfloor \frac{d}{2} \rfloor$th-order secure when implemented in software and executed by a CPU. To deal with the leakage of masked software, ILA violations first need to be identified and then eliminated.

The identification of ILA violations is usually done empirically by executing the masked software on a microprocessor. However, the results of this analysis are highly dependent on implementation details of the CPU, including placement,

routing, and the used standard cell library [Aro+21; MPW22]. Additionally, identifying the concrete root causes of leakage, e.g., components in the CPU, is a very challenging task, especially when working with CPUs that adapt a more complex microarchitecture. To eliminate ILA violations, changes to the masked software implementations need to be made. One strategy is to rewrite the software with respect to the identified leaking instructions and CPU components [PV17; She+21a; She+21b]. Another strategy is to use the *lazy engineering* approach [Bal+14], which tries to address this issue by using a higher protection order than theoretically required to compensate for the loss of masking order without identifying the concrete leakage sources. While both strategies result in a secure implementation, they come with very high overhead in terms of runtime and memory [Gro+16a; PV17].

Main Contributions In this thesis, we investigate the security of masked software implementations in practice when executed on CPUs. We explore methods to facilitate the identification and efficient elimination of ILA violations. Our analysis includes both small and more complex microprocessors, as well as masked software running in bare-metal mode or as a task within an OS.

In [Gig+21], we demonstrate that glitches and transitions in CPU components, such as the register file or the ALU, are possible root causes of ILA invalidations. We show that eliminating the leakage is more efficient and easier when done both on the hardware level (by integrating small changes into the CPU netlist) and on the software level (by applying a set of constraints). Furthermore, we create a list of requirements that need to be fulfilled by the SRAM block attached to the processor to enable secure loads and stores of shares. This analysis is done using the 32-bit open-source RISC-V Ibex core. After applying the hardware fixes, we obtain a *secured* version of the core, which guarantees that masked software can be executed without invalidating the ILA, given that the software adheres to the constraints. To identify the ILA invalidations in the first place and verify that our fixes provide the expected security level, we propose a new multi-level formal verification approach called Coco. Coco includes the gate-level netlist of the CPU, as well as the masked assembly implementation, allowing the identification of ILA invalidations in the CPU netlist during the execution of the masked software.

This work was published at *USENIX Security 2021* in collaboration with Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. The respective publication can be found in Chapter 3.

In [GPM21], we show that constructing secure and efficient masked software becomes more challenging for CPUs that have a more complex microarchitecture. Using Coco, we demonstrate that for such processors, glitches in the forwarding (bypass) logic that connects the CPU's pipeline registers can potentially lead

to leakage in the execution of masked software. To formalize our findings, we adapt the order-reduction theorem such that the number of pipeline stages and execution units is considered. Again, we show that ILA invalidations can be eliminated by modifying both the hardware and the software. When following the proposed software constraints, masked software can be executed securely but not always efficiently. Therefore, we suggest several implementation strategies and coding techniques that allow us to apply these constraints more efficiently.

This work was published at *ASIACRYPT 2021* in collaboration with Robert Primas and Stefan Mangard. The respective publication can be found in Chapter 4.

In [GPM23b], we provide the first analysis of masked software when executed as a task in an embedded OS instead of bare-metal mode. We show that OS-specific events, such as context switches, can potentially lead to ILA invalidations, causing leakage when executing the masked software. This leakage mainly stems from overwriting shares in memory and transitions on the register file and memory read and write ports that occur during a context switch. To fix these issues, we explore several efficient strategies to harden a context-switching routine of an OS against these leakage effects, while keeping the overhead for unmasked software minimal.

This work was published at *AsiaCCS 2023* in collaboration with Robert Primas and Stefan Mangard. The respective publication can be found in Chapter 5.

Other Contributions This section briefly describes publications in the area of software masking that the author worked on as a co-author during her PhD, but are not included in this thesis.

In [Blo+22], we propose power contracts between the masked software and the CPU, which define the exact leakage behavior of every instruction. When constructing masked software, it is then enough to check it within the power contract, which makes verification much faster and offers vendors the opportunity to release a power contract corresponding to their CPU instead of the complete CPU netlist. To demonstrate our approach, we model a power contract for the *secured* IBEX core and give a formal proof that the contract is complete.

The paper "Power Contracts: Provably Complete Power Leakage Models for Processors" [Blo+22] was published at *CCS 2022* in collaboration with Roderick Bloem, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas.

In [Gig+24b], we construct an efficient second-order masked software implementation of the Ascon cipher. The resulting implementation does not require any fresh randomness, and applies various other techniques to handle ILA violations caused by the CPU directly on the software level. We evaluate the implementation in terms of performance and security on 32-bit ARM and RISC-V processors.

Our work includes both an empirical assessment based on Welch's t-test and a formal study using Coco and the *secured* Ibex core.

The paper "Efficient Second-Order Masked Software Implementations of Ascon in Theory and Practice" [Gig+24b] is currently in submission and is a collaboration with Florian Mendel, Martin Schläffer, and Robert Primas.

1.1.2. Masking in Hardware

To deal with glitches and transitions causing ILA invalidations in masked hardware circuits, glitch-resistant masking schemes like Threshold Implementations (TI) [NRR06] and Domain-Oriented Masking (DOM) [GMK16] have been suggested. Such schemes are based on the idea of inserting glitch-stopping registers and refreshing masks during critical computations. Applying such measures requires additional registers, gates, and RNGs, which leads to a significant increase in the chip area and latency and a general decrease in the efficiency of an implementation. One promising technique to reduce the randomness consumption of masked hardware designs is the changing of the guards (COTG) technique [Dae17]. To refresh an intermediate value, COTG suggests using a share of another unrelated intermediate value instead of fresh randomness. Since its proposal, COTG has been successfully applied to various kinds of cryptographic algorithms, including the AES. However, regarding AES, most implementations focus on the first-order case [Ask+22; Sug19; WM18], or optimize exclusively for randomness, resulting in a higher latency [Bey+21; DSM22].

Besides efficiency, another important aspect of masking in hardware is to verify that the resulting implementation is secure in the presence of physical effects. To do so, a wide variety of automated formal verification tools have been proposed [Bar+19; Bel+22; Blo+18; KSM20] that allow to check if the ILA is invalidated for a given masked hardware design. While these tools are applicable in many different scenarios, the focus is clearly on Boolean masking and designs intended for ASICs.

Main Contributions In this thesis, we study ways to improve the efficiency of masked hardware implementations in terms of randomness and latency. We research new formal verification techniques, including one for arithmetic masking and one for FPGA platforms.

In [Gig+24a], we propose a second-order AES design protected by DOM, which provides a decent tradeoff between latency and randomness. In particular, we significantly reduce the required amount of fresh randomness per encryption while keeping a latency of 5 cycles per round. Reducing the randomness is achieved by the COTG technique, i.e., by sharing fresh randomness across the S-boxes and using shares of state bytes as randomness in other unrelated computations. The resulting implementation only requires 3 200 random bits per encryption (instead

of 20 800), and has an area of 117 kGE (instead of 176 kGE). To verify that our concept of reusing randomness is valid, we use a modified version of Coco, which can verify hardware implementations and perform empirical measurements of our design on an FPGA board.

This work was published at *CHES 2024* in collaboration with Franz Klug, Stefan Mangard, Florian Mendel, and Robert Primas. The respective publication can be found in Chapter 6.

In [GPM23a], we extend the formal verification approach proposed by Bloem et al. [Blo+18] to the domain of arithmetic masking. The resulting verification tool is capable of handling both arithmetic and Boolean masking of any order. For the first time, we formally analyze so-called A2B and B2A conversion algorithms, which are required to switch from the Boolean to the arithmetic domain and vice versa. In our analysis, we investigate a popular A2B/B2A algorithm implemented in hardware and show that glitches might compromise its security. Furthermore, we show that the approach can also be applied in the software domain, allowing us to report new findings of leakage caused, e.g., by register transitions.

This work was published at *ACNS 2023* in collaboration with Robert Primas and Stefan Mangard. The respective publication can be found in Chapter 7.

In [GPM24], we provide new insights on the security of masking schemes when implemented on FPGAs. We demonstrate that FPGA-specific optimizations performed during the synthesis process, which translates the HDL design to an FPGA configuration file, might introduce glitches invalidating the ILA. Consequently, even when adapting a glitch-resistant masking scheme, there is no guarantee that it is still glitch-resistant after FPGA synthesis. To detect such effects, we present Fenix, the first formal verification tool that operates directly on FPGA netlists and can handle any-order masked hardware implementations.

This work was published at *HOST 2024* in collaboration with Kevin Pretterhofer and Stefan Mangard. The respective publication can be found in Chapter 8.

Other Contributions This section briefly describes publications in the area of hardware masking the author worked on as a co-author during her PhD, but are not included in this thesis.

In [Cas+24], we construct a tool to automatically generate pipelined masked hardware implementations. Based on a Boolean equation describing the functionality of a cryptographic primitive, our tool generates a secure masked hardware implementation of the desired protection order from composable building blocks. The resulting designs have a low area consumption due to the elimination of synchronization registers in the pipeline, which is achieved by assigning computations to register stages in an optimized way such that registers can be removed or merged without breaking the ILA.

The paper "Compress: Generate Small and Fast Masked Pipelined Circuits" is currently in submission and is a collaboration with Gaëtan Cassiers, Stefan Mangard, Charles Momin, and Rishub Nagpal.

In [Nag+22], we propose a low-latency hardware masking scheme. The idea is to synchronize the propagation of shares such that glitches cannot lead to violations of the ILA by adapting a dual-rail encoding instead of registers. Our approach is generic and can be integrated into already existing masking schemes such as DOM. We show that it can be used to construct single-cycle glitch-resistant masked S-boxes. To validate the practicality and security of the proposed technique, we perform empirical verification on an FPGA and formal verification with Coco.

The paper "Riding the Waves Towards Generic Single-Cycle Masking in Hardware" was published at *CHES 2022* in collaboration with Rishub Nagpal, Robert Primas, and Stefan Mangard.

Familiarity breeds contempt.

Taylor Swift – Bejeweled

2 Background and State of the Art

This chapter provides the necessary background for this thesis. We start with a brief introduction to cryptography in Section 2.1. In practice, cryptographic concepts are implemented by cryptographic devices, which we cover in Section 2.2. Physical attacks represent a serious threat to the security of such devices. We give a brief overview of possible attacks, in particular Side-Channel Analysis (SCA) attacks like DPA, and discuss some more recent attack techniques in Section 2.3. To counteract SCA attacks, masking is one of the most popular countermeasures, which we describe in Section 2.4. To obtain a secure masked implementation, the ILA needs to be fulfilled, which is not always the case in practice. Section 2.5 presents common ILA violations observable in masked hardware and software implementations. To detect these violations, it is necessary to verify a masked implementation, either empirically or formally. We cover empirical verification in Section 2.6 and formal verification in Section 2.7. In the context of formal verification, we give a description of the most common adversary models, and give an overview of state-of-the-art automated formal verification tools.

2.1. Cryptography

Cryptography addresses the security of information exchanged between multiple parties [Sch96]. The goal of cryptography is to provide confidentiality, data integrity, and authentication of messages through cryptographic primitives such as encryption schemes, hash functions, and digital signature schemes [MOV96]. Encryption schemes provide the confidentiality of messages, which means that the message is concealed in a way such that it can only be understood by the intended receiver. Hash functions can provide message integrity, which refers to detecting whether the concealed message was altered by an unauthorized third party. Message authenticity refers to the sender and receiver identifying each

other when communicating, which can be achieved by digital signature schemes. All these cryptographic building blocks are based on the usage of keys, which represent a piece of information exchanged by the sender and receiver before communicating and then used to encrypt and decrypt messages. The security of cryptographic systems and their ability to provide properties like confidentiality, integrity, or authentication relies on the secrecy of the cryptographic key.

In general, based on the usage of keys, cryptographic primitives can be divided into symmetric and asymmetric primitives [MOV96]. In this thesis, we mostly focus on symmetric cryptography, although some of our contributions in the area of arithmetic masking can also be applied in the asymmetric domain (cf. Section 2.4.2, Chapter 7). Therefore, we describe both briefly in the next section but choose the symmetric domain as the basis for the rest of this chapter.

2.1.1. Symmetric and Asymmetric Cryptography

Both asymmetric and symmetric cryptography work with encryption schemes that allow the transformation of a message to a ciphertext using encryption/decryption keys. In the case of asymmetric cryptography, two different keys are used for encryption and decryption, while two identical keys are used for symmetric cryptography. In general, an encryption scheme specifies the message space $\mathcal{M}$, the key space $\mathcal{K}$, and the ciphertext space $\mathcal{C}$. To encrypt a message $m \in \mathcal{M}$ using the key $a \in \mathcal{K}$ with the encryption function E, one computes the ciphertext $c \in \mathcal{C}$ by $c = E_a(m)$. E needs to be bijective, such that it can be reversed and a unique plaintext message can be recovered for each ciphertext during the decryption process. Therefore, to decrypt a ciphertext c using a decryption key $b \in \mathcal{K}$, one computes $m = E_b^{-1}(c)$. For instance, in a two-party communication setting, the sender transmits c over an insecure channel to the receiver, who decrypts c to obtain the message m.

Symmetric Cryptography In symmetric cryptography, the encryption key is equal to the decryption key, i.e., $a = b$, which is kept secret. Before starting the communication, the sender and receiver need to exchange a in order to agree on a common key, which is done using dedicated mechanisms for key agreement or key exchange. Block ciphers are one of the most important elements in many symmetric-key systems, which break up a message into n-bit blocks and map each block to a n-bit ciphertext [MOV96]. The mapping function must provide confusion and diffusion, which means that the relationship between key and ciphertext is as complex as possible (confusion), and every ciphertext bit is influenced by every key bit (diffusion) [Sha49]. One method to achieve confusion and diffusion is to follow the structure of a Substitution-Permutation network, which provides confusion in the substitution layer through a non-linear function and diffusion in the permutation layer through a linear function. The Advanced Encryption Standard (AES) [DR02; Nat01] is one of the most prominent symmetric block ciphers based on the SP principle. It works with a

block size of 128 bits, an 8-bit S-box in the substitution layer, and the ShiftRows, AddRoundKey, and MixColumns functions in the permutation layer.

Asymmetric Cryptography Asymmetric cryptography keeps a public key a for encryption and a private key b for decryption, for which it holds that $a \neq b$ [DH76]. To start the communication, the receiver generates a key pair (a, b) and transmits a over a potentially insecure channel to the sender. The sender then uses the public key a to encrypt a message, which can, however, only be decrypted by the receiver who possesses the private key b. An adversary is able to observe a and c, which allows them to encrypt a message but not decrypt c because b is unknown. One of the most famous asymmetric encryption schemes is the Rivest-Shamir-Adleman (RSA) cryptosystem [RSA78].

Trapdoor one-way functions are the core of asymmetric schemes. They are easy to compute but hard to invert unless some trapdoor information (the private key b) is known. The integer factorization problem, as used in RSA, is a popular choice for the trapdoor function. It is based on the multiplication of two large prime numbers. While it is easy to compute the product, it is very hard to factorize the result and find out which numbers were initially multiplied unless one number is known. Integer factorization is believed to be intractable for very large prime numbers on modern computers, although it may be solved efficiently by quantum computers in the future [Sho94]. If, one day, quantum computers are powerful enough, they could be used to break many cryptographic schemes that are currently in use. Consequently, research in the area of post-quantum cryptography (PQC) emerged in recent years. PQC aims at finding cryptographic algorithms that are secure even in the presence of an adversary who has access to a quantum computer by using alternative trapdoor functions.

2.1.2. The Black-box Model

The main goal of symmetric cryptography is to provide ways to communicate securely under the assumption that the encryption key is secret. To quantify the security of a symmetric encryption scheme, a specific attack model is used which defines the abilities and limitations of the adversary. Traditionally, symmetric encryption schemes have been designed to be secure in the *black-box model*.

In this model, adversaries can make encryption queries by taking a plaintext, sending it to the encryption function, and retrieving the respective ciphertext [Aum17]. The encryption itself is a black box because the internal state of the cipher is unknown, and only the input and output are observable. Further refinements of the black-box model are possible, for example, depending on whether the adversary is allowed to choose the plaintext or the ciphertext or whether only a single or multiple plaintext-ciphertext-pairs are accessible.

Attacks in the black-box model generally include mathematical and statistical attacks like differential or linear cryptanalysis [BS90; BS91; Mat93], but also exhaustive search where the adversary tries out all possible keys. Attacks in

the black-box model can further be divided into multiple categories [Aum17; MOV96; Sch96]. Ciphertext-only attacks allow the adversary to only observe the ciphertext, but not the plaintext, while known-plaintext attacks give access to both. When performing a chosen-plaintext attack, the attacker can actively decide which plaintext should be encrypted. Chosen-ciphertext attacks allow the attacker to submit several ciphertexts and observe the decrypted plaintext.

2.2. Cryptographic Devices

To use a cryptographic algorithm in practice, an implementation of the algorithm is constructed. Devices that execute these implementations and store secret encryption keys are called cryptographic devices [MOP07]. In general, implementations can either be done in hardware or in software. Since this thesis deals with securing cryptographic implementations in hardware and software, we discuss the characteristics of both implementation platforms more in detail. In Section 2.2.1, we describe cryptographic hardware implementations and give more details specifically about ASICs and FPGAs. In Section 2.2.2, we introduce CPUs and embedded operating systems, which are used to execute cryptographic software implementations. Modern ASICs, FPGAs, and CPUs are usually based on CMOS technology, which we cover later in Section 2.3.3.

2.2.1. Cryptographic Hardware

Cryptographic hardware implementations are typically created from functional descriptions, often formulated in a hardware description language (HDL) like Verilog or VHDL, and then employed in an ASIC or FPGA design. ASICs are ICs customized for a particular task or application. They are heavily optimized and tailored towards the respective use case, which makes them the most powerful and performance-driven implementation platform. FPGAs represent reconfigurable logic circuits consisting of lookup tables (LUTs), that can be configured to represent an arbitrary logic function, connected by a programmable interconnect. Compared to ASICs, the big advantage of FPGAs is the possibility to update, reusability, and shorter time-to-market, albeit FPGAs are clearly less efficient and less powerful.

For both ASICs and FPGAs, the design steps to take from the HDL to the final circuit are similar. First, during synthesis, the design is transformed into a gate-level netlist, which essentially represents a graph, where the nodes are logic gates and the edges are wires. In the case of ASICs, the logic gates are defined according to a standard-cell library, while for FPGAs, the logic gates are LUTs. In the next step, the floorplanning, placement and routing of the design happens. This includes deciding on the exact location of logic gates on the chip, and establishing the wires between them such that specifications like the clock frequency are met. In the case of an ASIC, the design is then sent to a fab where

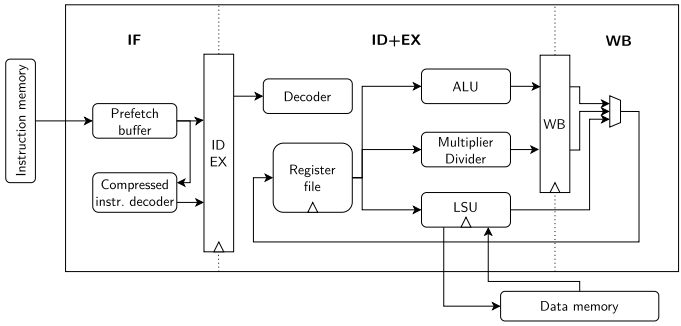


Figure 2.: Illustration of the 3-stage pipeline of the Ibex core (simplified), consisting of IF, ID+EX and WB stage [low24]

the manufacturing of the chip happens. In the case of an FPGA, a configuration file specifying the behavior and connection of LUTs is created and then flashed onto the FPGA.

Cryptographic hardware implementations are typically used as co-processors that are connected to a microprocessor and can be used to accelerate specific cryptographic operations. Depending on how tight or loose the connection (coupling) is, the co-processor and microprocessor can, for instance, communicate via a bus system or directly via custom instructions added to the processor's ISA. Numerous works describe the construction of co-processors for all kinds of cryptographic operations. For example, Steinegger et al. [SP20] propose a tightly-coupled accelerator for Ascon connected to a RISC-V CPU, that can be controlled via custom instructions added to the CPU. Another example is the work by Fritzmann et al. [Fri+22], who develop several side-channel protected PQC building blocks in hardware, where some can be accessed via an AXI bus, and some are directly integrated into the microprocessor. One famous example from outside of the embedded world is the AES accelerator built into many desktop-grade CPUs manufactured by Intel [Gue]. It can be controlled using dedicated instructions specified by the AES-NI instruction set. For instance, when executing the `AESENC` instruction, a whole round of an AES encryption is performed by a hardware circuit connected to the CPU. Besides co-processors, cryptographic hardware can also be found in other contexts, for instance, memory or disk encryption.

2.2.2. Cryptographic Software

As an alternative to hardware implementations, one can implement the cryptographic scheme in software and then use a general-purpose processor to execute the software. For embedded devices, compact microprocessors are used, rather

than large and powerful processors which can be found in desktop PCs and servers. Since embedded microprocessors deal with tasks that are less computing-intensive and often operate in low-power environments, they typically do not include more advanced features like out-of-order execution, complex branch prediction logic, virtual memory, or an extensive cache hierarchy. Such CPUs either directly run the software in bare-metal mode or as a task within an OS. Again, in the context of embedded systems, when referring to the OS we do not mean a full-scale Linux or Windows system, but rather a simple embedded OS, sometimes even providing real-time functionality. Code for cryptographic purposes is usually written directly in Assembly, or a high-level programming language suited for embedded use-cases like C, and then translated into Assembly with a compiler. The Assembly instructions are then executed by the CPU.

Microprocessors Microprocessors work by executing instructions that are located in some type of memory, such as a ROM or a RAM. The CPU first fetches an instruction, executes it, and writes the result back to memory. The *architecture* of a microprocessor refers to the Instruction-Set Architecture (ISA), which defines the instructions a processor can execute and provides a description of the configuration and data registers available [Pag09; Pec08]. Compared to that, the *microarchitecture* of a CPU refers to the concrete implementation of the ISA. Figure 2 shows an example of the microarchitecture of the RISC-V Ibex core. In the following, it serves as our example since other embedded processors follow a very similar structure. The Ibex core features a pipeline with three stages: Instruction Fetch (IF), Instruction Decode and Execution (ID+EX), and Writeback (WB). Other central building blocks are the register file, which consists of multiple general-purpose registers, several computation units, including the Arithmetic Logic Unit (ALU) and the multiplier, and the Load-Store Unit (LSU) to handle memory accesses.

Pipelining is used to increase the throughput of a processor, i.e., the number of instructions executed per time unit. The idea of pipelining is to split the execution of an instruction into multiple stages, where each pipeline stage deals with completing a specific step in the execution of an instruction [BO16; HH12; HP12]. Different pipeline stages complete different steps of different instructions in parallel. Therefore, in a pipeline with n stages, n different instructions can be processed in parallel, although each in a different level of completion. For example, the Ibex core can decode and execute the first instruction while fetching the second instruction in the same cycle.

Sometimes, it is not possible to execute all instructions in the pipeline in the given order, for example, when one instruction depends on the result of another instruction which has not yet finished its execution [BO16; HH12; HP12]. For instance, in the Ibex core, one instruction in the ID+EX stage might require the result of the previous instruction, which resides in the WB stage and has not yet been written into the register file. In such a situation, it might be necessary to

stall the pipeline, i.e., pause the execution of all previous instructions until the result of the required instruction becomes available in the register file. Clearly, stalls have a negative impact on the performance of the CPU. A more efficient alternative is forwarding. Forwarding logic, also called bypass logic, allows to send the result of an instruction from a later pipeline stage back to an earlier one if required [BO16; HH12; HP12]. Besides forwarding, many other concepts to optimize pipelining exist, e.g., superscalar execution. Superscalar processors employ copies of certain pipeline stages to execute multiple instructions in the same stage simultaneously. For instance, a processor could contain two ALUs, allowing it to compute the result of two `add` instructions in the same cycle.

Embedded Operating Systems Many embedded systems require multitasking functionality to handle the communication over a bus or the network, or to acquire and process sensor data. One common example are smart meters that record information about the energy consumption of a household while communicating with the grid operator and providing statistics to the local customer over WiFi. In such situations, dedicated embedded OSs are used that run the cryptographic software as one out of many concurrent tasks or processes. Unlike in desktop-grade Linux systems, the tasks that will be spawned by the OS are already known at compile time. This is especially important to be able to meet real-time guarantees if required by the application. Hence, the CPU executes the cryptographic software, which is compiled together with the embedded OS in bare-metal mode. Switching from one task to another is enabled by interrupts, which are triggered periodically by a timer or non-periodically by external events such as IO operations. On a software level, a task switch is called a context switch, where the register contents of one task are saved to memory, the scheduler selects the next task and the register contents of the next task are loaded from memory.

2.3. Physical Attacks

Cryptographic schemes have been designed to maintain security in the black-box model. To be used in practice, these schemes are implemented in hardware or software and then executed by cryptographic devices. In reality, attackers frequently manage to gain physical access to the device, which turns the black box into a gray box, paving the to physical attacks. Physical attacks exploit additional leakage caused by the fact that cryptographic devices run implementations of cryptographic schemes that violate the assumptions of the black-box model. Side-channel attacks are a subclass of physical attacks in which the attacker passively observes and analyzes the leakage (the side-channel information) emanating from the device. The device's power consumption is one powerful example of side-channel information that can be exploited in the context of a power analysis attack.

In the following, we give an overview of physical attacks in general (Section 2.3.1), then cover side-channel attacks (Section 2.3.2), and give more details about power analysis attacks (Section 2.3.3, Section 2.3.4). Throughout this and the following chapters, our explanations and discussions are based on the example of a cryptographic device running a symmetric encryption scheme. However, the statements are not limited to this setting and generally extend to other contexts as well.

2.3.1. Overview

The *gray-box model* assumes that the adversary has physical access or is in very close vicinity to the device when performing the encryption queries. Physical access allows the recording of physical characteristics, such as power consumption or timing, which can be analyzed to gain insight about the internal state of the cipher. Therefore, the encryption is not a black box anymore which shields any views from the adversary on intermediate computation results of the cipher. Instead, it is rather a gray box that allows observing some intermediate computation results besides the input and output when performing the encryption. Physical attacks have been shown to be very powerful since information on the internal state of the cipher can often be used to find out the secret encryption key, which leads to a complete break of the security of a system.

Mangard et al. [MOP07] propose a classification of physical attacks into passive and active attacks, which we briefly summarize in the following. When performing a passive attack, the adversary is merely observing the device and recording its physical properties while performing the encryption queries. By contrast, the goal of an active attack is to change the behavior of the device by tampering with its inputs or the execution environment. For instance, during a fault attack, the adversary operates the device outside the specification by increasing or decreasing the temperature or changing the clock speed or supply voltage for the purpose of skipping encryption rounds or creating a bias in the output ciphertext of multiple encryptions with the same input that eventually hints the key. Active attacks are out-of-scope for this thesis.

A further categorization of passive attacks according to the degree of invasiveness is possible, as discussed in [KK99; MOP07]. Invasive attacks completely disassemble the cryptographic device, for instance, by etching the chip's surface, which is extremely powerful but also requires rather expensive equipment. Non-invasive attacks are performed without changing the device, which is typically cheaper. Non-invasive, passive attacks are also known as side-channel attacks, which are the focus of this thesis.

2.3.2. Side-Channel Attacks

Side-channel attacks exploit the unintentional leakage of information via the physical properties of a device [And08; MOP07; SKS09; Spr+18; Sta10]. This

information is called side-channel information. During the attack, the adversary first records the side-channel information using dedicated measurement instruments and then tries to infer the secret key from the recorded information using statistical tools. Various kinds of physical properties have been used to extract secrets from cryptographic devices. The first side-channel attacks targeted the execution time of cryptographic implementations and were based on analyzing differences in the execution time caused by the input data and secret key [Koc96; OST06; SWT01]. One famous example is leaking the secret exponent in an RSA cryptosystem that is based on the square-and-multiply algorithm for modular exponentiation. It is based on the observation that if a bit of the secret exponent is 1, the square and the multiply function are executed, which takes longer than executing only the square function in case a bit is 0. Ever since the publication of these attacks, the constant-time property has become an essential requirement to secure cryptographic code.

Even if implementations are constant-time, there are still other kinds of side-channel information that can be exploited, including electromagnetic radiation [GMO01; Hey+12; QS01], sound [Gen+19; GST14], temperature [HS13] and photonic emission [FH08; Sch+12; Sch+13; Sko09]. One of the most commonly used side channels is power consumption [KJJ99]. To perform a power-analysis attack, an adversary uses an oscilloscope to measure the voltage drop over a shunt resistor placed in the VDD or GND path of the device. The voltage drop is proportional to the power consumption of the device, which differs depending on which data/key is used and can therefore be analyzed to be exploited.

2.3.3. CMOS Power Consumption

Today, almost all ICs are built using CMOS technology, mostly because of its low power consumption. The implementation of a CMOS cell is based on MOSFET transistors (NMOS and PMOS), which basically represent voltage-controlled switches. If the input signal is 1, NMOS transistors act like a switch that is *on*, while PMOS transistors act like a switch that is *off*. The CMOS technology combines NMOS and PMOS such that they work in a *complementary* fashion. Every CMOS cell consists of a PMOS pull-up network connecting the output to 1 (VDD) and an NMOS pull-down network connecting the output to 0 (GND). For any input pattern, exactly one network is switched on, while the other is switched off [Voi13; WH11].

The power consumption of a CMOS cell comprises a static and a dynamic part. Static power is consumed when the inputs are constant and no switching activity happens, and is mainly due to a small leakage current [AR02; Yea11]. It is negligible in most cases because in any stable state, no conductive path between VDD and GND exists. The static power consumption is by far outweighed by the dynamic power consumption, which occurs when the inputs of a CMOS cell change. The main reason for dynamic dissipation is the charging and discharging of load capacitances and the small short-circuit current which flows for a very

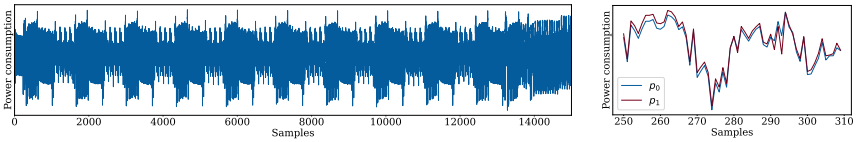


Figure 3.: Power consumption of an AES encryption. Left: Power trace of complete encryption. Right: Difference in the power consumption when processing plaintexts p_0 and p_1

short period of time when the transistors switch, and both the pull-up and pull-down network are conducting simultaneously [AR02; WH11; Yea11]. Therefore, whenever the input of the cell transitions from 0 to 1 or from 1 to 0, a different amount of power is drawn, especially compared to the case where the input does not change. Consequently, the dynamic power consumption is highly dependent on the processed data and the executed operations [AR02; Bak10; MOP07; WH11].

In Figure 3 on the left, we show an example of a power trace of a single AES encryption. One can clearly identify the 9 AES rounds, which are followed by the 10th shorter round, which does not compute MixColumns. Additionally, within each round, it is possible to see differences depending on which concrete operation (AddRoundKey, SubBytes, MixColumns, ShiftRows) is executed. On the right side, we plot the power consumption of the AES encryption for two different plaintexts p_0 and p_1 using the same key, and zoom into the beginning of the first round. The difference in the power consumption is clearly visible, underlining that it is dependent on the processed data but also dependent on the executed operations, as it can be seen on the left side.

2.3.4. Power Analysis Attacks

Power analysis attacks exploit both data and operation dependencies in the power consumption of cryptographic devices. Data dependencies are connected to the secret key and plaintext since cryptographic devices mostly process this kind of data. Operation dependencies can be exploited because although the adversary is not assumed to know the implementation, they typically know which cryptographic algorithm is executed. The first power analysis attacks, SPA (Simple Power Analysis) and DPA (Differential Power Analysis) [KJJ99], inherently targeted dynamic power consumption. More recently, successful attacks based on the static power consumption [MM21; MMR17; Moo19; Mor14; Poz+15], as well as the impedance of the chip [MMT23] have been demonstrated. They show that the static power consumption is also data-dependent because the amount of leakage current of a CMOS cell differs depending on the concrete input value and that the difference is significant enough to be exploitable in practice.

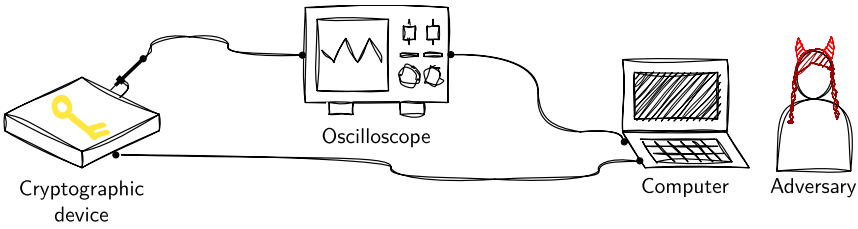


Figure 4.: Typical measurement setup for an SCA attack

For more details, we refer to the discussion in the respective papers.

The attention of this thesis is on attacks targeting dynamic power consumption. In the following, we first describe a typical setup to conduct a power analysis attack and give an overview of SPA, DPA, and state-of-the-art attack techniques.

Attack Setup One central part of any power analysis attack is measuring the power consumption of a cryptographic device. Figure 4 shows a typical measurement setup, which consists of the cryptographic device storing the secret key, an oscilloscope, and a computer operated by the adversary. The cryptographic device can be any CMOS circuit performing a cryptographic operation, e.g., an encryption, using a secret key. The computer is connected to the cryptographic device in order to perform encryption queries, which involves sending a plaintext to trigger the encryption and receiving a ciphertext. An oscilloscope is used to record the power consumption of the device by measuring the voltage drop over a shunt resistor placed in the VDD or GND path of the device with a probe. The oscilloscope returns the measured power consumption in the form of a power trace to the computer. A power trace, as it is shown on the left side of Figure 3, consists of multiple sampling points, each representing the power consumption of the circuit at a specific point in time. The remainder of the attack is executed by the adversary on the computer, which typically involves statistical analysis of the power traces.

Simple Power Analysis In the case of SPA, the attacker tries to disclose the secret key by looking at only a few, or even a single, power trace [KJJ99; MOP07]. SPA tries to exploit patterns in the power trace, which are caused by the key-dependent timing or order of operations or key-dependent processed data values. One famous example is attacking square-and-multiply in an RSA cryptosystem, which is based on a specific pattern in the power trace that can be observed depending on whether a bit of the secret exponent is 0 or 1. One possibility to improve SPA attacks are template attacks [CRR02], where the adversary first systematically characterizes the device's power consumption using templates and then uses the templates in the SPA attack. Soft-Analytical Side-Channel

Attacks (SASCA) represent a further improvement of template attacks based on belief propagation [VGS14]. Since their publication, SASCA attacks have been successfully demonstrated in various contexts [KPP20; Li+22; PP19; PPM17; You+23].

Differential Power Analysis DPA aims at disclosing the secret key by considering a large amount of power traces, usually a few thousand or even millions [KJJ99; MOP07]. The main idea is to exploit the differences in these power traces, which were recorded for multiple different inputs but with a constant key. DPA works in a divide-and-conquer fashion by splitting the encryption key into smaller chunks, which are then leaked individually. For example, in the case of AES-128, exhaustive search has an attack complexity of 2^{128}, while with DPA, it can be reduced to $256 \times 16 = 2^{12}$ by attacking the key byte-by-byte. In the following, we briefly summarize the steps of a classic DPA attack using the example of a single key byte of AES-128, based on the description of [MOP07]. For more details, we refer to [MOP07].

First, the adversary chooses an intermediate value for the attack that is computed during the encryption and depends on the key byte and input plaintext in a non-linear way. In the case of AES, the output of the SubBytes operation, $Sbox(k \oplus p)$ makes a good candidate. Then, the adversary records n power traces, each consisting of m samples, by sending n random plaintexts to the encryption device. Knowing which plaintexts were sent to the device, the adversary then computes the hypothetical intermediate values for each possible subkey value. In the case of AES, there are 256 possible values per subkey (subkey guesses). For each key guess k_g, and for each of the n plaintext bytes p_i, the hypothetical intermediate value is calculated by $v = Sbox(k_g \oplus p_i)$. By applying a power model, the hypothetical power consumption for each v can be computed. The power model is a function $f : \mathbb{R} \to \mathbb{R}$ that maps v to the approximate amount of power consumed to compute it. Popular choices are the Hamming weight (HW) model ($f(v) = HW(v)$), the Hamming distance (HD) model ($f(v, w) = HD(v, w) = HW(v \oplus w)$), the identify model ($f(v) = v$), the Least Significant Bit (LSB) model ($f(v) = v\&1$), or the Most Significant Bit (MSB) model ($f(v) = v\&128$).

Finally, the adversary matches the hypothetical power consumption for each subkey guess to the recorded power traces. The subkey guess, for which the hypothetical power consumption best matches the power traces, is most likely the correct key guess. As a measure of comparison, the Pearson correlation coefficient is frequently used. DPA attacks based on the correlation coefficient are also called Correlation Power Analysis (CPA).

Others Over the years, power analysis attacks evolved in many ways. For instance, adversaries have been becoming more powerful simply due to the availability of a better measurement setup, which allows them to record more power traces in a shorter amount of time. Another interesting research direction

is the combination of power analysis and machine learning, which is based on training a classifier with previously collected templates and using the trained model to perform the attack [Ben+20; HGG20; Hos+11; Ker+22; Kim+19; MDP20; MPP16; MWM21; Pic+19; Pic+23; Wan+23; Zai+20]. Similar to that, there also exist several works that utilize side-channel analysis attacks to leak the architecture of neural networks instead of keys from cryptographic devices [Bat+19; MBC21; Wei+18; Yos+20]. Furthermore, Remote Power Analysis attacks represent a class of power analysis attacks that do not require that the adversary has direct physical access to the cryptographic device or is even in close vicinity [MDB21]. Instead, such attacks use hardware components that are natively part of the cryptographic device as voltage sensors that can be read remotely by the attacker. For example, in a multi-tenant FPGA cloud scenario, the adversary can build a voltage sensor based on ring oscillators to spy on the victim running on the same FPGA [Gra+19]. The topic of remote power analysis has been analyzed by many other works [JUP24; Kra+19; Lip+21; OD19; Ram+18; Sch+18; Udu+22].

2.4. Masking against SCA

One of the most popular countermeasures against power analysis attacks is masking, which aims at decoupling the secret key from the data processed by the cryptographic device. In this section, we first explain the basic working principle of masking schemes (Section 2.4.1). Section 2.4.2 presents the different types of masking. In Section 2.4.3, we provide an overview of the most commonly used masked gadgets, which represent the basis of masked implementations. Later, the application of the masking countermeasure to cryptographic hardware or software implementations is discussed in Section 2.5, as well as methods to check if a masked implementation is secure (Section 2.6, Section 2.7).

2.4.1. Overview

Masking was initially proposed in 1999 by Chari et al. [Cha+99] and Goubin et al. [GP99], and is based on the principle of randomizing intermediate variables. An intermediate variable is sensitive if it depends both on the plaintext and the secret key. Randomization is established by splitting the sensitive intermediate variable into multiple random shares and adapting the cryptographic algorithm to process these shares instead. Consequently, the power consumption of the device depends on the shares instead of the sensitive intermediates, preventing the exploitation of side-channel leakage.

Given a sensitive intermediate value s, the core idea of masking is to split it into N random shares $s_1, ..., s_N$ using the splitting operation $\circ$ such that:

$$s = s_1 \circ s_2 \circ ... \circ s_N \tag{1}$$

The shares $s_1, ..., s_{N-1}$ are sampled randomly from a uniform distribution, while $s_N = s \circ s_1 \circ s_2 \circ ... \circ s_{N-1}$. The N-tuple $(s_1, ..., s_{N-1})$ is called a *sharing* of the sensitive variable s. Masking relies on the fact that an adversary who manages to recover $N-1$ shares cannot learn any information about s because any subset of up to $N-1$ shares is statistically independent of s. The masking type is determined by the splitting operation $\circ$, which is often the Boolean XOR ($\oplus$), the modular addition ($+$) or the modular multiplication ($*$). In practice, the generation of shares is performed by the cryptographic device, before the execution of the encryption algorithm. The sharing does not only affect the sensitive intermediate values but also the cryptographic functions which are applied to it. Any function F is split into multiple component functions $F_1, F_2, ..., F_N$ such that the combination of the outputs of these resemble the original function's output, i.e., $F(s) = F_1 \circ F_2 \circ ... \circ F_N$. In case F is a linear function, it can simply be called for every share individually, i.e., $F(s) = F(s_1) \circ ... \circ F(s_N)$. In case F is a non-linear function, the situation is more difficult since every component function operates on more than one share, which requires the correct addition of randomness to prevent the accidental unmasking of intermediate computation results. In the end, a masked implementation of an encryption algorithm outputs N shares, which need to be combined together by the cryptographic device to retrieve the actual ciphertext.

Masking is parameterizable by the security order d. In general, dth-order masking protects against dth-order DPA attacks, which exploit the joint leakage of d intermediate values [Din+14; MOP07]. Hence, to obtain a dth-order secure implementation, at least $d+1$ shares are required. Since handling more shares leads to a larger implementation overhead, working with the minimum number of shares ($N = d+1$) is the preferable option. For example, in Section 2.3.4, we run a first-order DPA attack by considering the leakage of only *one* intermediate value. Besides the independent leakage of shares, sufficient measurement noise is a crucial requirement for the security of masking. Masking of order d can only provide protection against dth-order attacks if the measurement samples are sufficiently noisy because then the number of required measurements grows exponentially with d [Cha+99]. Many works deal with the relation between noise and the security order d, e.g., [Bar+17; DFS15a; PR13; RPD09; Sta+10], as well as higher-order attacks in general [Cor+13; Gie+10; MM17; PRB09]. Formalizations in this direction are often done in adversary models, which we discuss in Section 2.7.1. More recently, several works use machine learning techniques to improve attacks on masking [GHO15; Lu+21; PWP22; Tim19; WPP20; Wu+23a]. Other than DPA, it has been shown that masking can have a positive effect on SPA but should generally better be combined with other countermeasures like shuffling [Jen+23; PPM17]. Several works have also pointed out that, in some cases, masking can be used to defeat fault attacks [ACS18; BH08; Dob+18; DOT24; MSS24].

2.4.2. Masking Types

A masking scheme is characterized by the choice of the splitting operation $\circ$.

Boolean Masking The most popular type of masking is Boolean masking, where the exclusive OR ($\oplus$) is used as the splitting operator. It is frequently applied to symmetric cryptographic algorithms that use linear operations and non-linear S-boxes over a characteristic-two field $\mathbb{F}_{2^k}$. For such algorithms, linear operations such as adding the round key with an XOR operation or bit permutations are easy to mask with Boolean masking, as they can just be applied to every share independently. For instance, to apply first-order Boolean masking to AES, the 128-bit state and the 128-bit key are first split into two 128-bit shares each. The linear functions of the AES (ShiftRows, MixColumns, and AddRoundKey) are applied to the shares of the state within the round function and the shares of the key within the key schedule individually. The AES S-box represents a non-linear function because $Sbox(s_1 \oplus s_2) \neq Sbox(s_1) \oplus Sbox(s_2)$, and therefore requires dedicated solutions that will be discussed later in terms of masked gadgets Section 2.4.3. In this thesis, we focus on Boolean masking, which we refer to in the following sections unless otherwise stated.

Arithmetic Masking Another type of masking is arithmetic masking, where the relation between shares of a sensitive value s is the modular addition: $s = \sum_i s_i = s_1 + ... + s_N \mod q$. The modulus is either a power of two ($q = 2^k$) or a prime number ($q \in \mathbb{F}_q$). One common use-case are ARX-based constructions like the hash function SHA-256 [Nat02], the stream cipher ChaCha [Ber08], or the block cipher Speck [Bea+13], where each round consists of a modular addition (using a power-of-two modulus), a rotation and an exclusive OR operation. In practice, masking these primitives requires both arithmetic masking (for the modular addition) and Boolean masking (for the rotation and exclusive OR). One option to deal with this is to stay in the Boolean domain and use a dedicated algorithm to securely add Boolean shares, as e.g. proposed by [CGV14]. Alternatively, one can switch between the arithmetic and Boolean domain using dedicated conversion algorithms called A2B and B2A [BCZ18; BDV21; CGV14; Cor+15; Cor+22; Cor17; Gou01; HT19]. The implementation of A2B and B2A algorithms in a secure way is challenging since they involve information from all shares but must not accidentally leak information about the unmasked value s.

With the rise of the PQC, arithmetic masking, especially using a prime modulus, has gained a significant amount of new attention. The fundamental building block of many PQC algorithms is polynomial multiplication, which can be implemented using the number theoretic transform (NTT), which is most efficiently masked in the arithmetic domain. At the same time, Boolean masking is required to protect building blocks like Gaussian samplers and decoders. Therefore, efficient conversion techniques for PQC have been researched more extensively in recent years, especially focusing on the use of a prime modulus [BC22; Fri+22; Sch+19].

Other Types Several other masking types have been proposed in literature. Multiplicative masking uses field multiplications $\otimes$ to protect sensitive intermediates by computing $s = \otimes_i s_i$ [GT02; MOP07; MQ]. It has, for example, been applied to mask the AES S-box, which represents a field inversion ($Sbox(s) = s^{-1}$) [AG01; MRB18]. Due to the close relation of Multi-party Computation (MPC) and masking, which are both secret sharing techniques, several proposals borrow ideas from the MPC domain [GSF14], for instance, masking based on Shamir's Secret Sharing [GM11a]. Inner product masking follows a slightly different approach, representing shares in the form of two random vectors that are connected by the inner product [Bal+12; Che+21; DF12].

2.4.3. Masked Gadgets

Masking cryptographic operations affects both the processed data, which is split into multiple shares, and also the operations, which need to be replaced by masked *gadgets*. For a function $F(s)$ that is computed by the unprotected implementation, a gadget $F'(s_1, ..., s_N)$ is used to replace it in order to obtain the masked implementation. Gadgets implement exactly the same functionality as the original function, i.e., if the Boolean masked gadget $F'(s_1, ..., s_N)$ produces the output sharing $(s'_1, ..., s'_N)$ then $s'_1 \oplus ... \oplus s'_N = F(s)$. Gadgets are used both in hardware and software, although some gadgets are more optimized for either use case. For instance, to build a masked hardware implementation, the logic gates in the unprotected circuit are replaced by suitable gadgets. For masked software, an implementation technique called bit-slicing is very popular. Bit-slicing reduces the computation of a function to bitwise logic operations (for example AND, XOR, OR, NOT) by utilizing an arrangement of the bits of an input data word in the CPU registers [AP21; GR16; Kön08; MN07], allowing to execute several instances of the function in parallel. To obtain a masked implementation, these bitwise instructions are replaced by masked gadgets consisting of (multiple) instructions.

Masked gadgets can be divided into affine gadgets, refresh gadgets, and multiplication gadgets. Affine gadgets are primarily used to mask exclusive OR computations and negations and can simply be constructed by applying the original function to every share independently because, e.g., in the first-order case $F(s) = F(s_1) \oplus F(s_2) \oplus c$ for a constant c. For example, the addition of the round constant Rc of the AES to a state byte s is represented by the function $AddRC(s, RC) = s \oplus RC$. The masked version is then $AddRC(s_1, RC) \oplus AddRC(s_2, RC) \oplus c$ with $c = RC$. Refresh gadgets are used to re-randomize a sharing for a sensitive variable, which can improve the security of a masked implementation. They essentially represent the identity function, i.e., $F(s) = s$, but transform the input sharing $(s_1, ..., s_N)$ into a fresh output sharing $(s'_1, ..., s'_N)$ by setting $s'_i = s_i \oplus r_i$ for $1 \leq i \leq N-1$, and $s'_N = s_N \oplus_i r_i$.

Multiplication gadgets are a fundamental building block to implement non-linear functions, such as masked S-boxes, by providing a way to multiply two shared field elements. A one-bit multiplier corresponds to an AND-gate, which is

why multiplication gadgets are also called masked AND gadgets. The construction of secure masked multipliers is not trivial and requires the careful addition of fresh randomness. In a first-order Boolean masking scheme working with 1-bit values, a multiplication gadget combines the value u, represented by the sharing (u_1, u_2), with the value v, represented by the sharing (v_1, v_2), such that for the resulting value w, represented by the sharing (w_1, w_2), it holds that $w = u \wedge v = (u_1 \oplus u_2) \wedge (v_1 \oplus v_2)$. Naively, one can compute the shares (w_1, w_2) using the distributive property:

$$w_1 = (u_1 \wedge v_1 \oplus u_1 \wedge v_2) \tag{2}$$

$$w_2 = (u_2 \wedge v_1 \oplus u_2 \wedge v_2) \tag{3}$$

However, this is not secure because w_1 is not independent of v, and w_2 is not independent of u. Over the years, many works have proposed constructions to compute the product of two shared variables securely. In the following, we will discuss the most frequently used masked gadgets in more detail.

Trichina AND Gate Trichina et al. [Tri03] proposed one of the first masked multipliers, which is suitable for first-order Boolean masking. It introduces a fresh random value r that is used to construct the shares of w as follows:

$$w_1 = r \tag{4}$$

$$w_2 = (((r \oplus u_1 \wedge v_1) \oplus u_1 \wedge v_2) \oplus u_2 \wedge v_1) \oplus u_2 \wedge v_2 \tag{5}$$

The Trichina AND gate provides first-order security given that operations are applied in the exact order as depicted above, i.e., the random value needs to be added to the first partial product in the beginning. Several masked implementations have been constructed using the Trichina AND gate, especially to mask the AES S-box [Bal+15; SS16; Tri03].

Treshold Implementations (TI) Threshold Implementations (TI) [NRR06] do not exclusively focus on multiplication gadgets but represent a more general framework for the design of masked implementations. The main idea of TI is to decompose a cryptographic algorithm into multiple component functions that fulfill the correctness, non-completeness, and uniformity properties. Correctness means that the sum of the outputs of the component functions resembles the same result as the original, unmasked function. Non-completeness states that every component function must only operate on a subset of the input shares. Uniformity refers to the distribution of input and output shares, which should be uniform to prevent problems when composing multiple component functions. Implementing a first-order secure multiplication gadget requires at least three shares to fulfill the non-completeness property. For instance, one possibility to

implement such a multiplier is:

$$w_1 = F_1(u_1, u_2, v_1, v_2) = u_1 \wedge v_1 \oplus u_1 \wedge v_2 \oplus u_2 \wedge v_1 \oplus r_0 \oplus r_1 \quad (6)$$

$$w_2 = F_2(u_1, u_3, v_1, v_3) = u_3 \wedge v_3 \oplus u_1 \wedge v_3 \oplus u_3 \wedge v_1 \oplus r_1 \quad (7)$$

$$w_3 = F_3(u_2, u_3, v_2, v_3) = u_2 \wedge v_2 \oplus u_2 \wedge v_3 \oplus u_3 \wedge v_2 \oplus r_0 \quad (8)$$

In this example, every component function excludes at least one of the three shares. Uniformity is achieved by adding the fresh random variables r_0 and r_1. TI, in principle, comes with a larger overhead due to the fact that more than the minimum number of shares is required, although this often allows us to avoid using fresh randomness. TI has been popular for hardware implementations because it is provably secure against glitches (cf. Section 2.5.1), and numerous works applying the scheme to various kinds of algorithms exist, e.g., [Bil+13; Bil+14; Caf+21; Cha+22; Gro+15; Jat+20; Mor+11; STE15]. Furthermore, masked software implementations based on TI have been proposed [Cha+22; GD23; SBM18; She+21a].

Ishai-Sahai-Wagner (ISW) Multiplier Masking schemes that are generic in terms of the protection order do not only defend against first-order DPA, such as Trichina's AND gate but rather support an arbitrary protection order d. One of the first works in this direction is the ISW multiplier proposed by Ishai et al. [ISW03]. The multiplication of the Boolean sharings $(u_1, ..., u_{d+1})$ with $(v_1, ..., v_{d+1})$ results in the output sharing $(w_1, ...w_{d+1})$. The multiplication is performed in two steps. First, the random bits r_{ij} for $1 \leq i < j \leq (d+1)$ are generated. Second, the output shares w_i for $1 \leq i \leq (d+1)$ are computed by:

$$w_i = u_i \wedge v_i \oplus \big(\bigoplus_{i \neq j} (r_{ij} \oplus u_i \wedge v_j) \oplus u_j \wedge v_i \big) \quad (9)$$

Based on the ISW multiplier, Rivain et al. [RP10] propose a generic masking scheme for AES together with a security proof. Furthermore, it is a popular construction to build conversion algorithms to convert between Boolean and arithmetic masking, as shown by several works [BCZ18; Cor+15; Cor17].

Domain-Oriented Masking (DOM) Another generic masking approach is DOM, which was presented by Gross et al. [GMK16]. The idea of DOM is to assign each share a domain, and to keep the domains independent from each other. Domains are similar to the component functions in TI but less restrictive such that it is possible to work with the minimum number of shares ($d+1$ shares for security order d). A DOM multiplication gadget is structured in three phases: calculation, resharing, and integration. In the calculation phase, the shares are multiplied in pairs, resulting in inner-domain $(u_i \wedge v_i)$ and cross-domain $(u_i \wedge v_j, i \neq j)$ terms. Cross-domain terms are then reshared using fresh randomness in the resharing

phase. Due to the resharing, the cross-domain terms are statistically independent of other values and, therefore, can be summarized into domains again in the integration phase. In a dth-order DOM masking scheme, one output share of the multiplier w_i for $1 \leq i \leq (d+1)$ is given by:

$$w_i = \overbrace{u_i \wedge v_i}^{\text{Calculation}} \oplus \underbrace{\Big(\bigoplus_{j>i} \overbrace{u_i \wedge v_j}^{\text{Calculation}} \oplus r_{i+j(j-1)/2} \Big)}_{\text{Refreshing}} \oplus \underbrace{\Big(\bigoplus_{j<i} \overbrace{u_i \wedge v_j}^{\text{Calculation}} \oplus r_{j+i(i-1)/2} \Big)}_{\text{Refreshing}} \quad \text{Integration} \tag{10}$$

We mark the terms computed in the calculation and refreshing phase with blue and red braces, respectively. The integration phase includes the addition of all three partial sums. Compared to ISW, DOM comes with significant advantages in terms of masked hardware implementations, such as shorter delay paths by balancing the arrangement of multiplication terms and improved security in the presence of glitches (cf. Section 2.5.1).

Nowadays, several projects make use of DOM for side-channel hardening. For example, Google launched the OpenTitan project [Joh+18; low19a; low19b], a commercial-grad open-source hardware root of trust, which employs a first-order DOM-masked AES implementation. Besides that, implementations of Ascon [Gro; Pra+23], Keccak [GSM17], and AES [GMK17] based on DOM have been proposed. Gross et al. [Gro+16b] propose a DOM-masked RISC-V processor that can execute unprotected software implementations in a side-channel protected manner. Kiaei et al. [KS20] and Marshall et al. [MP21] suggest an ISA for masked software implementation using DOM. Furthermore, Fritzmann et al. [Fri+22] use DOM to build a masked adder for Boolean shares. DOM is even applied in the area of side-channel resistant machine learning [Dub+22] and as part of a combined countermeasure against fault attacks [Gru+21].

Composable Multiplication Gadgets To securely mask a cryptographic algorithm as a whole, several masked gadgets need to be stacked together. However, just because the security of a masked gadget was proven, it does not imply that the *composition* of several of these gadgets is also secure. This does not only hold for the composition of multiplication gadgets, but can even lead to insecure designs when composing an affine (linear) and a non-linear gadget. Therefore, deriving composability properties for masked gadgets is currently an important research topic [Bar+15; Cas+21; CS20; KSM20], also because it is not always feasible to (formally) verify the security of a complete cipher due to complexity. Instead, one could prove the security and composability of a small masked gadget and follow a bottom-up approach to construct the complete masked design. One of the biggest challenges when designing composable gadgets is to keep the overhead

low since composability is enabled by frequent refreshing, leading to increased randomness consumption.

2.5. Masking in Practice

Masking is a provably secure countermeasure. Its security against SCA can be formally proven in theory with respect to a set of assumptions. One of these assumptions is the independent leakage assumption (ILA) [Ren+11], which states that the shares are always leaked independently from each other. It has been shown that the ILA does not always hold, leading to a gap between the theoretical and practical security of masking. More concretely, to use a masking scheme in practice, it needs to be manifested in an implementation. One option is to implement the masking scheme in hardware, for instance, as an ASIC or on an FPGA. For masked hardware implementations, physical side-effects of CMOS circuits, like glitches, have been shown to break the ILA. Section 2.5.1 focuses on these effects with respect to masked hardware implementations in detail. Alternatively, one can craft the masked implementation in software, which can then be executed on a CPU. CPUs are usually also based on CMOS technology, which also exposes them to these physical effects that may compromise the security of masked software. Section 2.5.2 explains the connection between physical effects and masked software in detail.

To be practical, masking schemes need not only to be secure but also efficient. While masking already comes with a relatively large overhead, dealing with ILA violations further drastically increases the cost. Therefore, methods to decrease the overhead by optimizing masked implementations are important to make masking countermeasures more practical. In Section 2.5.3, we discuss state-of-the-art optimization techniques for both hardware and software implementations.

2.5.1. ILA Breaches in HW

CMOS circuits are comprised of combinatorial subcircuits consisting of logic gates and registers. The input signals for each gate usually do not arrive at the same time because of different gate delays and wire lengths. The gate delay, or propagation delay, is the time a gate needs to "react" to input changes, i.e., the time it takes to produce an output in response to a change of its inputs [KL03; WH11]. It depends, among other things, on the type of gate and the concrete input values. Given that in combinatorial logic, multiple logic gates are cascaded, the individual arrival time of an input signal at a gate is determined by a variety of factors, including the delay of all previous gates that were passed through and the individual wire lengths. The gates might temporarily produce an incorrect output signal, e.g., when one of the input signals has already arrived but the other one has not, until both input signals have reached their stable state and the correct result is computed. Such temporary effects are called glitches.

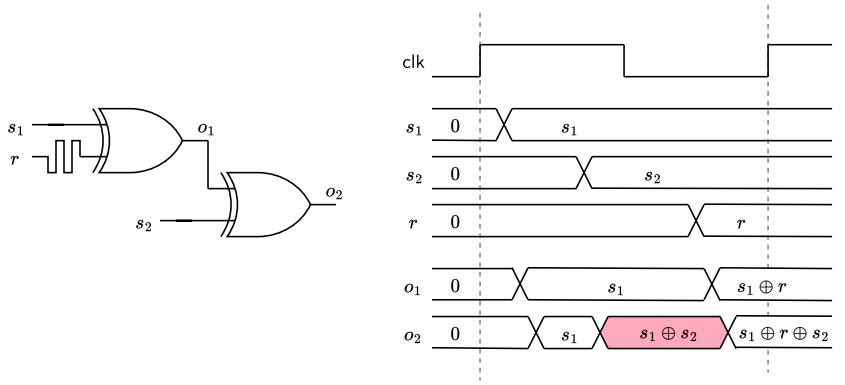


Figure 5.: Example of a first-order masked circuit computing $(s_1 \oplus r) \oplus s_2$. The sensitive value s is represented by the sharing (s_1, s_2), and the value r represents fresh randomness. Due to glitches, the second XOR gate temporarily computes $s_1 \oplus s_2 = s$, leading to an insecure design.

Many works point out that the security of masked hardware circuits is compromised by glitches [FG05; GMK16; ISW03; Moo+19; MPG05; MPO05; Rep+15]. In a CMOS circuit, glitches lead to unexpected combinations of data values for a short period of time until all signals are stable. In a masked CMOS circuit, data values are shares, and combining these shares leads to the non-independent leakage of them, which breaks the ILA. Figure 5 shows an example of a first-order masked circuit, working with a sharing of the sensitive variable $s = (s_1, s_2)$ and the fresh random value r. It computes $(s_1 \oplus r) \oplus s_2$, which is valid from a masking perspective. Additionally, the partial sum $(s_1 \oplus r)$ is valid and independent from s. The circuit consists of two XOR gates, where $o_1 = (s_1 \oplus r)$ and o_2 computes the final result. In the example, the wire lengths between the inputs s_1 and s_2 and the respective XOR gates are short, while the wire length between the input r and the XOR gate is long. Therefore, s_1 will arrive at the second XOR gate before being combined with r. Additionally, s_2 will arrive at the second XOR gate quickly, and o_2 temporarily computes $s_1 \oplus s_2 = s$, which refers to a leak in the masking scheme. After a while, r will also arrive and propagate through the circuit, o_1 computes the final (stable) result, $s_1 \oplus r$, and o_2 computes $s_1 \oplus r \oplus s_2$, which is secure.

This example highlights very well the gap between theory and practice in masked design. In theory, the computation is secure because no intermediate computation result depends on the sensitive value. Even the designer who constructs the implementation, e.g., in Verilog, can hardly see any problem. The issue arises for the first time after the HDL model has been processed by the synthesis, placement, and routing flow because the decisive factors (wire lengths)

are picked in that stage. At the same time, other placement strategies might lead to shorter wire lengths between r and the XOR gate, which causes r to always arrive first and does not create any issues. Note that in this example, we assume uniform gate delays and extreme setup times to better highlight the problem.

Transitions typically occur when data from the previous cycle which still resides in the circuit at the beginning of the current clock cycle. When the data from the current cycle propagates through the circuit, it is possible to observe a transition from the old to the new data value. In the context of masking, this can, in the worst-case, cause transitions from one share to another, and the attacker can observe the Hamming distance between the first and the second share, effectively breaking the ILA. One typical example of leakage caused by transitions is overwriting a register that stores one share with its counterpart. Transitions in masked hardware implementations are addressed by many recent works [Cor+12; CS21; Dho21; Mül+22].

A third effect that breaks the ILA in masked CMOS circuits is coupling, which basically refers to crosstalk between adjacent wires and the physical proximity of shares [Cnu+17; Dho21; Gur+23; LBS19; SK23]. For example, De Cnudde et al. [Cnu+17] show that masked TI designs are vulnerable due to coupling between shares on an FPGA. Levi et al. [Gur+23; LBS19] show in several experiments that coupling-based side-channel leakage can be amplified by the adversary through tweaks applied to the measurement setup. In the future, the challenge posed by coupling in the context of masked designs will become even more relevant. With the ongoing evolution of CMOS technology, distances between wires on the chip will shrink more and more, which provokes crosstalk.

Glitches, transitions, and couplings have been analyzed for ASICs, but also specifically for FPGAs [Cnu+17; GLE15; Li+20; MM12; Mül+23; Roy+15]. One example is the work of Roy et al. [Roy+15], who study an implementation of a first-order masked SIMON without synchronization and show that glitches lead to leakage when implemented on an FPGA.

Countermeasures As illustrated in the example, glitches are difficult to predict when designing masked hardware implementations. The first proposals tried to balance or reorder the operations in a circuit, such that the glitches causing ILA breaches can be eliminated [Ala+09; Gho+07; KMC07]. Nowadays, solutions have been shifted to the algorithmic level because they are easier to apply for more complex circuits and do not require modifications to the back-end design flow. The foundation of algorithmic defenses are registers, which stop glitches from propagating and, therefore, serve as synchronization points. To make the circuit in Figure 5 secure, a register would need to be inserted between the two XOR gates, storing the value of o_1. As a result, the register ensures that only $s_1 \oplus r$ is forwarded to the second XOR gate, but never s_1 alone, independent of how fast r reaches the first XOR gate.

Glitch-resistant masking schemes are defined on algorithmic level when registers

need to be used in order to prevent glitches. TI [NRR06] was the first provably secure glitch-resistant masking scheme. Due to the non-completeness property, no glitch in a component function can reveal any information about all shares of a sensitive variable. In order to ensure that the property also holds when the output of a component function is used as the input of the next, the result needs to be stored in a register. DOM [GMK16] uses registers to secure crossings of share domains. To ensure that the refreshing phase is completed before the integration phase, a register is placed after combining the fresh randomness with the cross-domain terms. It prevents that due to a glitch, multiple cross- or inner-domain terms are combined, which results in statistical dependence of the sensitive value, and breaks the ILA. Composable masking schemes provide another way to build glitch-resistant designs by composing multiple, smaller glitch-resistant building blocks [Cas+21; Cas+24; Fel+22; Kni+22; Mül+23].

Transition effects receive slightly less attention in the context of masked hardware, as defeating them can basically be done by the careful handling of registers. For instance, masked encryption schemes are often implemented in a pipelined fashion, where the encryption state is stored in registers connected by combinatorial logic. Every share is stored in its own "copy" of the state registers (in its own domain), which is updated with a share of the same domain after the round function has been computed.

2.5.2. ILA Breaches in SW

It is challenging to maintain the theoretical protection order of masked software implementations in practice when executed by a CPU. The typical reason for the observed leakage are transitions occurring in the CPU microarchitecture, causing distance-based instead of value-based leakage [Bal+14]. While value-based leakage means that an implementation leaks the computed intermediates individually (hence, corresponds to the ILA), distance-based leakage describes that intermediates could also be leaked in pairs (hence, violating the ILA). For instance, overwriting a register that stores value a with the new value b exhibits distance-based leakage of $HD(a, b) = HW(a) \oplus HW(b)$, that is, a transition from a to b. Papagiannopoulos et al. [PV17] call this effect the *register overwrite effect*, or short, the *overwrite effect*. Register overwrite effects can easily be introduced by compilers when writing masked software in a higher-level language such as C because the register allocation is done by the compiler, which has no intuition about masking. Even when masked software implementations are written in Assembly, it is often challenging to prevent register overwrite effects since the exact expressions stored in registers need to be tracked by the programmer. Before overwriting the register with the value from another register or from memory, it needs to be ensured that the transition between the old and new value does not violate the ILA. If that is not possible, another register needs to be chosen, or the register first needs to be cleared. Several works provide empirical evidence of the register overwrite effect in practice, including [Bal+14; Bec+22;

MMT20; PV17; She+21b]. The overwrite effect can also be observed for data memory, i.e., when overwriting a share stored in memory with another share of the same sensitive variable [Cor+12; PV17].

While one could argue that register overwrite effects can be detected by carefully investigating the Assembly code, other leakage effects are more difficult and sometimes even infeasible to see. The reason is that they often reside from the microarchitecture of the processor, which is, in most cases, closed-source and not known by the designer. Several works discuss the leakage caused by microarchitectural elements of the processor [Gig+21; GPM21; MPW22; PV17; She+21a; She+21b]. For example, Papagiannopoulos et al. [PV17] point out the neighbor leakage effect, which refers to distance-based leakage observed between two distinct data storage units, e.g., registers, although an instruction accesses only one of them. Related to that, data manipulated by two distinct instructions might be leaked via transitions in hidden microarchitectural storage elements, such as registers used by the memory bus [She+21b]. Gao et al. [Gao+20] show that it is even possible that bits stored in the same registers are leaked when performing a bitwise instruction, e.g., by barrel shifters in the ALU.

Countermeasures One option to address leakage in masked software implementations is to simply ignore the problem. This approach is called *lazy engineering*, and is based on the order reduction theorem proposed by Balasch et al. [Bal+12]. The theorem states that a masking scheme that is dth-order secure assuming value-based leakage (intermediates are leaked individually) is $\lfloor \frac{d}{2} \rfloor$th-order secure assuming distance-based leakage (intermediates are leaked in pairs). As a consequence, a designer applying the lazy engineering approach would design a dth-order secure masked software implementation if $\lfloor \frac{d}{2} \rfloor$ is practically required. The advantage of lazy engineering is that it can be applied without knowledge about the microprocessor. The disadvantage is, however, the large overhead: for instance, to get 2nd-order security in practice, a 4th-order implementation needs to be constructed.

Another option is to (empirically) build a leakage model that characterizes the leakage behavior of the target microprocessor, and then adapt the masked assembly implementation in a way such that it is secure in the leakage model. Adaptations might require programming tricks, such as inserting dummy memory accesses to clear registers of the memory bus, temporarily using additional randomness to mask data stored in registers, or rotating shares before storing them in registers [Bar+21a; PV17; She+21b].

A quite different approach is to modify the processor such that SCA-protected operations are facilitated [CPW24; Gao+21; Gro+16b; MGH19; SS22; TKS11]. For example, Cheng et al. [CPW24] propose an Instruction-Set Extension (ISE) for RISC-V that contains dedicated instruction for masked software implementations. However, such approaches often imply a large overhead for the CPU as well as a certain performance penalty with respect to (unprotected) general-purpose

software.

2.5.3. Optimizing Masked Implementations

Applying the masking countermeasure to the implementation of a cryptographic algorithm comes at a significant cost. On the one hand, this cost originates from the algorithmic changes to enable the masking countermeasure in the first place. This includes splitting the sensitive intermediates into multiple shares (which potentially requires additional RNGs for mask generation), and processing them by multiple instances of linear functions, or dedicated non-linear functions respectively. On the other hand, a certain amount of overhead stems from compensating for ILA breaches, such as defeating glitches and transitions.

Optimizing Masked Software Typical cost metrics to quantify the overhead of masked software are the amount of memory (RAM) that is needed on top, the code size, and the runtime overhead, usually measured in clock cycles. Existing optimization strategies often combine bit-slicing with clever register scheduling to minimize memory accesses, which can be expensive in terms of CPU cycles [GR17; SS16]. De Groot et al. [Gro+16a] propose an optimized implementation of the PRESENT cipher on an ARM Cortex-M4, that tries to lower the overhead caused by lazy engineering with bitslicing.

Optimizing Masked Hardware The overhead of a masked hardware implementation is usually evaluated in terms of area, latency, and randomness. Latency refers to the (minimum) number of clock cycles required to run an implementation. The randomness includes both offline and online randomness. Offline randomness is used to obtain the initial sharing of the input, while online randomness is consumed by the design during the encryption, e.g., for refreshing. However, randomness overhead will eventually translate into area because when requiring more randomness, more/larger/more powerful RNGs are needed to deliver the random bits, and additional logic gates are needed to add them to the masked design. While techniques like DOM, TI, or composable masking schemes brought secure masking into practice, the security came with a certain cost, which ever since has been tried to be minimized by the scientific community. In the context of optimizations, there clearly exists a tradeoff between latency and randomness/area. For example, glitch-stopping registers increase the latency of a design. Eliminating such registers requires adding more fresh randomness at another point of the design, which increases the randomness/area.

Several proposals following either of the two optimization directions (reduction of latency and reduction of randomness/area) exist. Various proposals in the direction of low-latency masking exist [AZN21; GIB18; KM22; Nag+22; Sas+20; Sim+22; Sim+23]. For example, the low-latency variant of DOM [GIB18] is based on the idea of eliminating the glitch-stopping register by skipping the

compression step in the masked multiplier and duplicating the necessary logic respectively. Removing registers also has a negative impact on the length of the critical path, which affects the frequency at which the design can run in practice. While a non-optimized first-order masked AES S-box implementation runs in 8 cycles and requires 18 bits of fresh randomness, their low-latency variant runs in a single cycle but requires 2 048 bits of fresh randomness. In order to reduce the amount of randomness required in a design, one method is using a freshly generated random value at several independent points in a design, or by applying the changing of the guards (COTG) technique. Works in this direction include [Bey+21; Fel+22; KM23; Pap18]. COTG was initially introduced by Daemen [Dae17] to achieve uniformity in TI-based designs more efficiently. This work showed that instead of freshly generating a new random value, it is simply possible to use an unrelated share of the cipher state, which is independent of the one being refreshed. Since its proposal, COTG has been successfully applied to various kinds of cryptographic algorithms, e.g., [ANR19; Bey+21; Gig+24a; JPS18; SD17; Sug19; WM18].

2.6. Empirical Verification of Masking

After creating a masked implementation, designers need to test if the security order in practice adheres to the theoretical protection order by checking if running the implementation exhibits any observable side-channel leakage. Leakage can simply be caused by implementation errors (bugs), or by ILA breaches which were not sufficiently handled. Empiric verification of masked implementations involves the collection of power traces of the cryptographic implementation and the subsequent analysis of these traces to identify leakage.

2.6.1. Collecting Power Traces

Masked hardware implementations are subject to empirical verification at multiple stages throughout the design process. Clearly, it is desirable to detect potential issues as early as possible. To facilitate this, FPGA boards are commonly utilized for initial power measurements of the masked design alongside simulation tools. Masked software implementations can often directly be run on the target microprocessor, or a simulator for the respective device is used. In the following we focus on how power traces were collected for the designs in this thesis. We refer to the work of Buhan et al. [Buh+22] for a broader overview and comparison of automated leakage detection tools.

Power Traces from Concrete Devices Masked hardware implementations are frequently assessed on an FPGA board to collect power traces in the early stages of the design process. In this thesis, we work with different FPGA evaluation boards that are specifically designed for SCA evaluations, the SAKURA-G

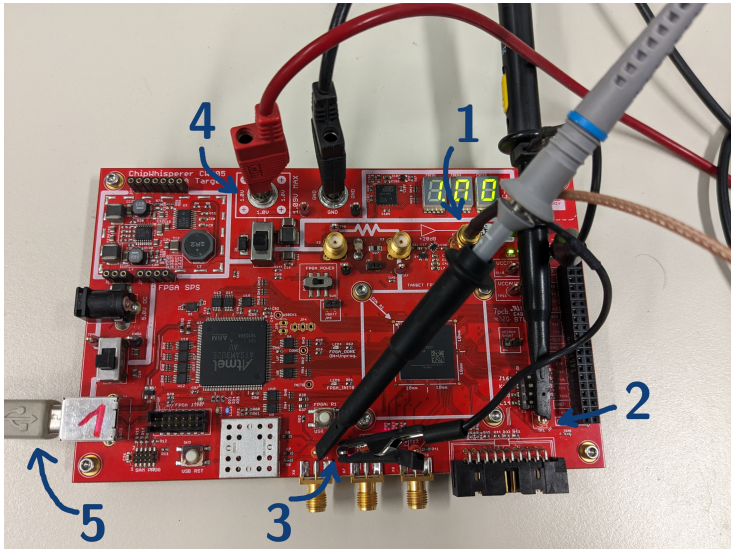


Figure 6.: Measurement setup based on the CW305 evaluation board used in this thesis, consisting of: (1) connection to oscilloscope to measure the power consumption, (2) trigger signal, (3) clock output, (4) external power supply, (5) USB connection to PC to provide plaintexts

board [GIS14], and the NewAE CW305 board [New24]. Figure 6 shows our measurement setup to acquire power traces based on the CW305 evaluation board. The cryptographic implementation which we want to assess runs on the FPGA. The power consumption is measured via a cable connected to the oscilloscope (1). In order to determine when the cryptographic operation starts, we use a trigger signal (2). To improve the quality of measurements, we synchronize the clock signals of the FPGA and oscilloscope (3) and use an external, low-noise power supply (4). Finally, to provide the implementations with different plain texts, we connect our lab PC via USB (5). To perform post-silicon evaluations, the ASIC chip can, for example, be embedded into a PCB [MOP07].

Masked software implementations can directly be run on the microcontroller, which is usually also embedded into a PCB to facilitate power measurements. The setup is in general very similar to the one described above, the main difference is that the cryptographic implementation is not run on the FPGA, but on the microprocessor.

Leakage Simulators Estimating the power consumption via simulations represents an alternative to collecting power traces directly from a concrete device. Simulations can be performed in cases where no measurement equipment is

available, extremely low-noise power traces are needed, or when only a part of the design should be assessed instead of the complete implementation. Given a masked hardware circuit, one simple way to approximate the dynamic power consumption is to count how many signal transitions happen on a gate's output during one clock cycle while simulating the circuit [TV05]. Another more recent approach in this context is `PROLEAD` [MM22], which does not approximate the dynamic power consumption of a circuit directly but instead simulates the values of intermediate variables. Based on this simulation, the tool later runs statistical tests to detect leakage in masked implementations.

For masked software implementations, leakage simulators follow a different path since the power consumption of microprocessors without access to the microarchitecture is hard to estimate, especially when considering the effects of breaking the ILA. Instead, the goal is more to build a leakage model of the processor which determines the leakage behavior of each instruction on the CPU. Designers of masked software can use the resulting leakage model to estimate the security of their implementation. For example, ELMO [MOW17] was built based on an empirical analysis of the power consumption of instructions executed on the ARM Cortex-M0 processor. ASCOLD [PV17] can be used to assess implementations for the 8-bit ATMega163 microcontroller. As an extension to `PROLEAD`, `PROLEAD-SW` [ZMM23] has been proposed that allows to simulate and analyze the leakage of masked software running on an ARM CPU.

2.6.2. Analyzing Power Traces

In order to perform empirical verification of a masked implementation, the power traces obtained by either physical measurements or simulations are statistically analyzed to detect potential leakage. One option is to run a DPA attack, as described in Section 2.3.4, which simply allows to learn whether the implementation withstands this specific attack or not. No conclusions about all the other attack models and approaches that have been proposed over the years are possible. Another option is to run a TVLA (Test Vector Leakage Assessment), as described by Goodwill et al. [Goo+11], which aims at uncovering statistical dependencies between the power consumption and processed data in general. TVLA comes in several different variants, as proposed in [Goo+11]. In this thesis, we focus on the non-specific fixed vs. random test since this is also the most common in literature.

TVLA is based on Welch's t-test, which measures the significance of the difference of means of two distributions. In the context of cryptographic implementations, it compares the leakage of a cryptographic device while processing a fixed plaintext to the leakage while processing a random plaintext. In both cases, the same encryption key is used. The idea of the t-test is that if the power consumption of the fixed and the random group can be distinguished, it is data-dependent and can, therefore, potentially be exploited by attacks like DPA that are based on differences in the power consumption.

To perform a t-test for a cryptographic device, the device is first programmed with a fixed key that remains unchanged throughout the experiment. Then, the power consumption of the device is recorded while processing either a random or a fixed input plaintext. The traces are are assigned to the sets $S_f \in \mathbb{R}^{n_f \times m}$ (fixed plaintexts) or $S_r \in \mathbb{R}^{n_r \times m}$ (random plaintexts). n_f denotes the number of traces recorded for a fixed plaintext, while n_r denotes the number of traces recorded for a random plaintext. m is the number of samples a power trace consists of. Based on S_f and S_r, a t-statistic, or t-score $t \in \mathbb{R}^m$, is computed for each of the m samples:

$$t = \frac{\mu_f - \mu_r}{\sqrt{\frac{s_f^2}{n_f} + \frac{s_r^2}{n_r}}} \tag{11}$$

In this case, μ_f and μ_r are the means of S_f and S_r, and s_f and s_r are the standard deviations. The null hypothesis is that S_f and S_r have equal means, i.e., they cannot be distinguished. The null hypothesis is accepted if the t-score stays between ± 4.5, which means that it is not possible to say whether a fixed or a random plaintext was used based on the power consumption. If the t-score exceeds ± 4.5, the null hypothesis is rejected with a confidence greater than 99.999 %, which means that the power consumption shows data-dependent differences.

TVLA is frequently used to assess masked implementations. In that case, having a "fixed" key means choosing a value k for the key but refreshing the shares of k with every invocation of the cryptographic device. The shares of the fixed plaintext are handled accordingly, while the shares of the random plaintexts are simply chosen randomly every time. TVLA, as described above, can be used to detect first-order leakages in implementations, as the analysis is based on comparing the means (first statistical moment) of S_f and S_r. To detect higher-order leakages, it is necessary to investigate higher statistical moments, e.g., the variance for a second-order implementation.

The interpretation of the results obtained from TVLA needs to be done cautiously, as discussed in [Pap+23; SM15; Sta18]. If the t-score exceeds the ± 4.5 border, it just means there are data-dependent differences in the power consumption. It does not imply that these differences can be exploited successfully, e.g., in a DPA attack, nor does it tell anything about the attack effort in case an attack applies. For example, the t-score of a masked cryptographic implementation that loads the unmasked plaintext from memory will exceed the critical border because this causes data-dependent differences in the power consumption. However, it is not possible to do, e.g., a DPA attack, because this operation only involves the plaintext but not the secret key. On the other hand, if the t-score does not exceed the ± 4.5 border, it just means that there are no data-dependent differences in the power consumption. It does not prove that the implementation is secure, as it might still be broken in another evaluation setup, e.g., when using more traces, another test device, or another oscilloscope.

2.7. Formal Verification of Masking

Empirically verifying a masking scheme is very important for investigating the security of an implementation, but it is strongly bound to the used device, implementation details, and measurement equipment. For instance, a masked hardware implementation might be secure on an FPGA once, but when running the placement and routing process again, another placement of components on the same FPGA could lead to an insecure design. Furthermore, when not adjusted carefully, the lab setup could easily result in low-quality measurements, which means that for every empirical verification, a non-negligible amount of time must be spent on adjusting the equipment.

Formal verification tries to overcome these disadvantages by constructing a security proof for the SCA resistance of a given masking scheme. This allows making general security statements in a specific attacker model, clearly stating the adversary's abilities, and providing more independence of the concrete attack setup. Depending on the attacker model, it is also possible to include physical effects such as glitches and transitions. We compare and describe the most common attacker models in Section 2.7.1. Security proofs for masked circuits can also be automated by formal verification tools that read in the circuit and check if it is secure against dth-order attacks. Section 2.7.2 gives an overview of state-of-the-art tools. One possibility to implement a masking verification tool is to use Fourier-based verification, which is also applied by several works in this thesis [Gig+21; GPM23a; GPM24]. Therefore, we cover the necessary theoretical background in Section 2.7.3.

2.7.1. Adversary Models

Formal adversary models define the abilities of a side-channel attacker, allowing to argue more formally about the security of a masked implementation. They work with a circuit-based representation of the masking scheme, where the circuit is represented as a graph consisting of vertices and edges [CS21; ISW03]. The vertices are either the inputs or outputs of the circuit, logic gates, or registers, each processing elements from $\mathbb{F}_2$. Edges are the wires carrying elements from one logic gate to another.

Classic/Robust Probing Model The *d-probing model* was introduced by Ishai et al. [ISW03] in 2003. It is also known as the *standard probing model*, the *classic probing model*, or simply the *probing model*, and is still one of the most used models today. It states that the adversary possesses d probes that can be placed on any set of wires in the circuit to record the information carried by that wire for an infinite amount of time. A masked circuit provides d-th order security if the adversary cannot learn anything about any sensitive variable by combining the recorded observations. Later, Faust et al. [Fau+18] extended the classic probing model and proposed the *robust probing model* to include glitches, transitions

and coupling effects. In the robust probing model, the adversary works with (g, t, c)-extended probes, which optionally also allow to capture combinations of intermediate variables caused by glitches ($g = 1$), transitions ($t = 1$) or coupling ($c = 1$).

For example, consider the circuit in Figure 5 computing $(s_1 \oplus r) \oplus s_2$. This circuit is 1st-order secure in the probing model using $(0, 0, 0)$-extended probes. A 1st-order attacker is allowed to place one probe, which can either be placed on an input wire or the wires o_1 and o_2. Probing the input wires is not useful for the adversary. Probing o_1 will record $\{s_1 \oplus r\}$, which is independent of s. Probing o_2 will record $\{s_1 \oplus r \oplus s_2\}$, which is also independent of s. The situation is different for when working with (1,0,0)-extended probes capturing glitches, because probing o_2 records $\{0, s_1, s_2, r, s_1 \oplus r, s_1 \oplus s_2, r \oplus s_2, s_1 \oplus s_2 \oplus r\}$, and the term $s_1 \oplus s_2$ is not independent of s.

Noisy Leakage Model In practice, side-channel measurements are noisy. Consequently, other than suggested by the classic probing model, the adversary does not have direct access to the plain intermediates, allowing them to directly probe an intermediate X. Instead, they probe a noisy function of X, that is, $\nu(X) = X + \delta$, where δ follows a Gaussian distribution. This was first formalized in the *noisy leakage model* by Chari et al. [Cha+99] in 1999. They formally study the effectiveness of masking based on this model and show that the number of traces required to recover the sensitive value increases exponentially with the masking order d. One downside of the noisy leakage model is that formal proofs are not straightforward to obtain as rely on complex information-theoretic computations. Compared to the robust probing model, the noisy leakage model is strongly based on the ILA and, therefore, does not consider physical side effects like glitches. While the classic/robust probing model restricts the number of probes to d, and the noisy leakage model assumes jointly leaking wires, it has been proven that the classic probing model implies the noisy leakage model [DDF14].

Random Probing Model The *random probing model* states that every wire of a circuit leaks with a given probability p [Bel+20a; DDF14; ISW03]. Hence, an adversary placing a probe on a wire can only record the intermediate value with probability p and will not observe any leakage otherwise. p depends on the noise level, i.e., the higher the noise, the lower the probability that the adversary can observe the intermediate value. This allows us to further compute the expected number of traces required by the adversary to recover the sensitive variable s. Security proofs in the random probing model are considered more intuitive than in the noisy probing model due to the higher level of abstraction. It has been shown that the noisy leakage model reduces to the random probing model, but also that the random probing model reduces to the classic probing model [DFS15b]. Just like the noisy probing model, the random probing model is based on the ILA and, thus, does not consider glitches or transitions.

Adversary Models for Masked Software The classic probing model, noisy leakage model and random probing model are relatively high in abstraction such that they can be used to analyze the security of masked implementations under the assumption of independent leakage. The robust probing model captures physical effects that might violate the ILA but is formulated in terms of masked circuits and does not characterize the leakage of masked software implementations when executed by CPUs very well. For example, the robust probing model assumes that the adversary can probe a wire for an infinite amount of time, which is a valid assumption in the case of, e.g., a pipelined masked hardware circuit. When transferring this to a CPU, an attacker could choose to probe the read port of the register file, allowing them to access every intermediate value that is ever computed by the masked software. By combining these values, masked software implementations of any order could be broken.

A first work in the direction of adversary models for masked software is the proposal of distance-based leakage [Bal+14], which suggests that pairs of intermediates may be leaked by software implementations to represent the register overwrite effect. However, this does not capture transitions between internal CPU registers or glitches. The *register probing model* [Bel+20b] states that since software works by manipulating CPU registers, it is rather unrealistic that only a single bit of an intermediate is leaked. Instead, it is more likely, that all the bits of a register are leaked together (for example in the ALU), and can be observed by a single adversary probe. The register probing model does not capture the register overwrite effect or glitches. Barthe et al. [Bar+21b] suggest to build a CPU leakage model by explicitly characterizing the leakage of every possible instruction. While this approach yields a very accurate adversary model and allows the inclusion of various effects like the neighbor leakage effect [PV17], it does not consider glitches and is closely related to a specific CPU. In this thesis, we address the issue of adversary models for masked software in [Gig+21].

2.7.2. Automated Formal Verification

Security proofs for masking schemes can be done by using a pen-and-paper approach. With the growing circuit size and complexity, as well as the increasing protection order d, constructing a security proof manually becomes relatively cumbersome, especially considering glitches and transitions. Over the last years, several tools to formally verify masked implementations in an automated fashion have been proposed, covering a variety of adversary models. For a detailed overview and comparison of these tools, we refer to the work of Feldtkeller et al. [FSG23]. In the following, we want to focus on the techniques applied by those tools that are used to verify masked hardware implementations in the robust probing model. In general, to check whether a dth-order masked hardware implementation is dth-order probing secure, many formal verification tools take the gate-level netlist of the implementation as an input and either produce a security proof or point out the respective wires that need to be probed by the

adversary in order to recover the sensitive value.

maskVerif One of the first tools to implement an automated formal verification approach was maskVerif [Bar+15; Bar+19], which computes symbolic leakage sets for every operation performed by the masked circuit. The leakage set contains all intermediate computation terms with respect to the circuit inputs, which can be captured by an adversary by probing the gate output. To check dth-order security, any combination of d leakage sets is investigated with respect to its dependency on the sensitive value. maskVerif employs several optimizations to achieve better performance, mainly targeting the complexity of symbolic expressions. Since symbolic expressions are formulated based on the circuit inputs, the tool applies simplifications that remove a certain circuit input from the expression as long as the probe's distribution is not affected. These approximations lead to *non-completeness* because it might be assumed that attackers can probe more than what is practically possible, leading to false positives. Over the years, maskVerif was continuously improved, especially with respect to the number of supported adversary models, and its performance [Bar+16; Bar+17; Bar+19; Bar+20].

Rebecca In 2018, Rebecca was published by Bloem et al. [Blo+18] in order to formally verify any-order masked hardware circuits in the robust probing model. Similar to maskVerif, the tool computes a leakage set for every gate, which describes the arithmetic expression that can be probed by the attacker. Instead of deriving the leakage sets symbolically, Rebecca uses estimations based on the Fourier expansions of Boolean functions [ODo14], which allows to represent the output of a gate as a multilinear polynomial over the circuit input variables. If the coefficient of a linear combination of input variables in the polynomial is non-zero, it means that the gate output *correlates* with the respective input combinations. Rebecca checks for leaks by searching for gate outputs which correlate with all shares of a sensitive variable. For higher-order masking, combinations of leakage sets (correlation sets) need to be checked, which is accelerated by a SAT solver. Since correlation sets are estimated, Rebecca also represents a non-complete verification approach. In the next section, we will give more details on Fourier-based verification because some works of this thesis make use of it.

Silver Quite a different path is followed by Silver [KSM20; Mül+22], which computes the joint output distributions of each gate precisely and then performs the independence check. The exhaustive computation and analysis of probability distributions is computationally very expensive, and therefore, the tool relies on binary decision diagrams to speed up the verification. Silver supports a variety of different adversary models, including the classic/robust probing model, and can also verify composability properties of masked circuits. While Silver offers a complete verification approach (false positives are not possible), it eventually comes at the cost of efficiency and is limited to the verification of smaller gadgets.

IronMask In 2022, Belaïd et al. propose IronMask [Bel+22], which aims at a complete verification approach that is still efficient. Similar to maskVerif, the leakage sets are computed symbolically, but it offers a better algorithm for simplifying the expressions such that no false positives can occur. Consequently, IronMask is, in many cases, much faster than other complete verification approaches, especially when verifying higher-order security properties of masked gadgets or when verifying gadgets with non-linear randomness (gadgets where randomness is not added linearly, but by performing quadratic operations to mix input shares and randomness input shares). Besides (robust) probing security and compositional security notions, it also supports the random probing model, which is neither addressed by Silver nor by maskVerif.

2.7.3. Fourier-Based Verification

Any formal verification method needs to perform statistical dependency checks to determine if probing a certain expression reveals anything about the sensitive value. Rebecca utilizes the Fourier expansions of Boolean functions [BCG13; ODo14], which is closely related to statistical dependence. Consider a Boolean function $f(X) : \{-1, 1\}^n \rightarrow \{-1, 1\}$ over a set of input variables $X = (x_1, x_2, ...x_n)$, where -1 represents true and 1 represents false. The Fourier expansion of f represents it as a multilinear polynomial, that is, the sum of linear combinations of input variables with respect to a specific Fourier coefficient $\widehat{f}$:

$$f(X) = \sum_{T \subseteq X} \widehat{f}(T) \prod_{x_i \in T} x_i \tag{12}$$

The correlation of a Boolean function with regard to its inputs can be read off directly from the Fourier expansion. The Fourier coefficients tell the strength and type (positive or negative) of the correlation, while the respective linear combination includes the variables to which the linear dependency exists. More formally, a Boolean function does not correlate with $T \subseteq X$ iff $\forall T' \subseteq T$ it holds that $\widehat{f}(T') = 0$ [XM88]. For example, the Fourier expansion of an AND function $f(X) = a \wedge b$ for the input set $X = (a, b)$ considers the linear combinations $\{\{a, b\}, \{a\}, \{b\}, \{\}\}$, and is given by: $f(X) = -0.5ab + 0.5a + 0.5b + 0.5$. This means that the function correlates positively with a and b, has a constant bias, and correlates negatively with $a \oplus b$.

The inputs of a first-order Boolean masked circuit are either the shares of a sensitive variable s_1 and s_2, fresh random values, or any public values such as round constants. The gates in the circuit represent Boolean functions, performing some computations with respect to the circuit inputs. In order to formally verify that the circuit is secure, it is necessary to check if any gate correlates to $s_1 \oplus s_2$. This is done by computing the Fourier expansion of each gate with respect to the circuit inputs and verifying that the Fourier coefficient of $s_1 s_2$ is non-zero. For dth-order verification, it is necessary to test the nonlinear combination of

any tuple of d gates.

Computing the exact value of the Fourier coefficients is computationally expensive, and also unnecessary because it is sufficient to know if correlation exists, and not how strong it is or whether it is negative or positive. Therefore, Rebecca does not work with the exact Fourier representations, but with *correlation sets* instead, which contain all linear combinations of circuit inputs a gate correlates to. A correlation set $\mathcal{C}$ of a gate g computing a function $f(X)$ is described by the following condition:

$$\text{For } T \subseteq X : \prod_{x_i \in T} x_i \in \mathcal{C}(g) \text{ if } \widehat{f}(T) \neq 0 \tag{13}$$

For instance, the correlation set of the AND gate from before is then $\mathcal{C}(g) = \{\{a, b\}, \{a\}, \{b\}, \{\}\}$, which refers to the variables an attacker would be able to probe on the gate output.

Correlation sets can easily be used to create a verification approach that can deal with glitches. To do so, the attacker's abilities are extended such that it is possible to replace any gate in the circuit with a gate that computes an arbitrary Boolean function, while keeping the original gate's inputs. Additionally, even when assuming glitches, the correlation set of any register must only correspond to the stable value computed in the previous cycle. For example, consider again the circuit in Figure 5. Without glitches, the correlation set assigned to o_1 is $\mathcal{C}(o_1) = \{\{s_1, r\}\}$, while the correlation set assigned to o_2 is $\mathcal{C}(o_2) = \{\{s_1, s_2, r\}\}$. The circuit is first-order probing secure because the set $\{s_1, s_2\}$ cannot be probed in any scenario. However, with glitches, the attacker may replace both XOR gates by an AND gate, resulting in $\mathcal{C}(o_1) = \{\{\}, \{s_1\}, \{r\}, \{s_1, r\}\}$ and $\mathcal{C}(o_2) = \{\{\}, \{s_1\}, \{r\}, \{s_2\}, \{s_1, r\}, \{s_1, s_2\}, \{s_1, s_2, r\}\}$, which contains $\{s_1, s_2\}$, and hence, correlates to $s_1 \oplus s_2 = s$. To make the circuit secure, o_1 needs to be stored in a register in order to "stop" the glitch. The correlation set of o_1 would then stay the same, but the correlation set of the register output would be $\{\{s_1, r\}\}$, which changes the correlation set of o_2 to $\{\{\}, \{s_2\}, \{s_1, r\}, \{s_1, s_2, r\}\}$, which is first-order secure.

Part II.

Publications

3

Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs

Publication Data. Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *USENIX Security Symposium.* 2021.

Contribution. In the context of the Ibex core, the author of this thesis contributed to the identification of problems, the suggestion and integration of fixes as well as the secure SRAM model. Furthermore, the author of this thesis created the masked Assembly implementations, used them to evaluate the formal verification approach, performed the empirical verification on the FPGA, and did the area evaluations. The author of this thesis contributed to the formal verification concept. Finally, the author of this thesis significantly contributed to the written part of the publication.

Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs

Barbara Gigerl[1], Vedad Hadzic[1], Robert Primas[1], Stefan Mangard[1,2], Roderick Bloem[1]

[1] Graz University of Technology [2] Lamarr Security Research

Abstract The protection of cryptographic implementations against power analysis attacks is of critical importance for many applications in embedded systems. The typical approach of protecting against these attacks is to implement algorithmic countermeasures, like masking. However, implementing these countermeasures in a secure and correct manner is challenging. Masking schemes require the independent processing of secret shares, which is a property that is often violated by CPU microarchitectures in practice. In order to write leakage-free code, the typical approach in practice is to iteratively explore instruction sequences and to empirically verify whether there is leakage caused by the hardware for this instruction sequence or not. Clearly, this approach is neither efficient, nor does it lead to rigorous security statements.

In this paper, we overcome the current situation and present the first approach for co-design and co-verification of masked software implementations on CPUs. First, we present Coco, a tool that allows us to provide security proofs at the gate-level for the execution of a masked software implementation on a concrete CPU. Using Coco, we analyze the popular 32-bit RISC-V Ibex core, identify all design aspects that violate the security of our tested masked software implementations and perform corrections, mostly in hardware. The resulting *secured* Ibex core has an area overhead around 10%, the runtime of software on this core is largely unaffected, and the formal verification with Coco of an, e.g., first-order masked Keccak S-box running on the secured Ibex core takes around 156 seconds. To demonstrate the effectiveness of our suggested design modifications, we perform practical leakage assessments using an FPGA evaluation board.

1. Introduction

Since the rise of the Internet of Things (IoT), embedded devices are integrated into a wide range of everyday services. Often, these simple devices are part of larger software ecosystems, which makes the protection of cryptographic keys on these devices an essential but challenging task. Physical side-channel attacks, such as power analysis, allow attackers to extract cryptographic keys by observing a device's power consumption [CRR02; KJJ99; QS01]. To prevent such attacks, embedded devices typically employ dedicated countermeasures on the algorithmic

level. The most prominent example of such algorithmic countermeasures against power analysis is masking, essentially a secret sharing technique that splits input and intermediate variables of cryptographic computations into $d+1$ random shares such that the observation of up to d shares does not reveal any information about their corresponding native value [Bar+17; Bel+17; Cnu+16; GM17; GMK16; ISW03; Rep+15].

Masking schemes typically have in common that they rely on certain assumptions such as independence of leakage, i.e., independent computations result in independent leakage [Ren+11]. However, as pointed out by many academic works in the past, such assumptions are typically not satisfied on CPUs. Coron et al. [Cor+12] were among the first who showed that, e.g., memory transitions in the register file or RAM can leak the Hamming distance between two shares, thereby reducing the protection order of masking schemes on CPUs. Later publications follow up on these observations [Gro+16a; MMT20; PV17], and amongst others, formulate the so-called order reduction theorem [Bal+14]. This theorem states that dth-order protection under the assumption of independent leakage reduces to $\lfloor \frac{d}{2} \rfloor$-th protection if effects like memory transitions are taken into account. Consequently, and without further assumptions on the hardware, achieving second-order protection using masked software implementations can require computations with up to 5 shares.

This is a very significant overhead, and also the reason why the goal in practice is to find strategies to cope with the leakage caused by the underlying CPUs and to achieve dth-order protection with $d+1$ random shares. In order to test if such implementations indeed provide the desired security level in practice, research on the verification of masked cryptographic implementations has gained a lot of attention during the last years. The existing works can be roughly divided into two sets: works based on empirical verification, and works based on formal verification.

On the empirical side, authors have studied masking-related side effects of certain microprocessors via leakage assessments and then built corresponding hardened software implementations [Gro+16a; PV17]. While their resulting masked implementations do in fact maintain their theoretical protection in practice, they also come with a noticeable performance overhead (by up to a factor of 15) that is caused by the necessary software tweaks. Since leakage assessments are quite labor-intensive, tools like PINPAS [Har+03], or more recently, ELMO [MOW17] have been developed that can emulate power leakage for certain microprocessors. The authors of ROSITA [She+21] have pushed this automation even further by also automating the software patching process after leakage detection. A quite different take on providing side-channel protection on CPUs is presented by Gross et al. [Gro+16b], who propose a masked CPU design that can perform unprotected software implementations in a side-channel protected manner. Similar work exists for RISC-V processors [MGH19], also on instruction set architecture level [Gao+21; KS20; Reg+09].

On the formal side, tools like Rebecca [Blo+18] and maskVerif [Bar+19] represent the first steps toward formal verification of masked implementations. Both tools are mainly tailored to hardware implementations; maskVerif does offer some support for software implementations but (1) can only deal with code that is written in a special intermediate language, and (2) uses a probing model that only considers simple CPU side-effects such as register overwrites. More recently, Belaid et al. presented *Tornado* [Bel+20], a compiler that automatically generates masked software implementations that are secure in the same model. A more fine-grained software verification approach that utilizes annotated assembly implementations is presented by Barthe et al. [Bar+21], while with Silver [KSM20], Knichel et al. promise improved verification accuracy and performance for hardware implementations.

Our Contribution So far, the verification of masked software implementations was only done in simplified settings that require modified software implementations and do not consider a wider range of side-effects, such as glitches at the gate level, that occur when software runs on an actual CPU. There still exists a noticeable gap between correctness proofs and the resulting practical protection for masked software implementations. We close this gap by providing the following contributions:

- We present Coco, a tool inspired by Rebecca, that can formally verify the security of (any-order) masked, RISC-V assembly implementations that are executed on concrete CPUs defined by gate-level netlists. Coco essentially provides hardware-level verification including glitches for software implementations with constant control flow.
- Using Coco, we analyze the design of the popular 32-bit Ibex[1] core and identify all hardware design aspects that could prevent the leakage-free execution of our test suite of masked software implementations on this CPU.
- Based on this analysis, we present design strategies for CPU and memory, that with low hardware overhead, eliminate most of our discovered flaws in hardware, while leaving behind a few select and easy-to-check constraints for masked software implementations.
- We show the practicality of this work by verifying a variety of masked assembly implementations, including various types of (higher-order) masked And-gates, a second-order masked Keccak S-box [GSM17], and a first-order masked AES S-box implementation [BP12]. We also show examples where Coco identifies flaws in broken masked software implementations and reports the corresponding execution cycle, as well as the location of the

[1] https://github.com/lowRISC/ibex

leakage source within the Ibex netlist. To show the effective robustness of our secured design, we perform leakage assessments on an FPGA evaluation board.

- We publish Coco and our secured Ibex on Github[2].

Outline In Section 2, we present Coco, a tool that can formally verify the leakage-free execution of masked software implementations directly on CPU netlists. Section 3 explains how we analyze the popular 32-bit RISC-V Ibex core using Coco, the discovered issues, and the resulting hardware modifications which enable leakage-free software execution. In a similar spirit, Section 4 takes a look at data memory and proposes solutions for how SRAM can be added to a CPU core such that it can be included in Coco's verification. Section 5 describes Coco's verification workflow in detail and presents various verification runtime benchmarks as well as the practical evaluation. We conclude our work in Section 6.

2. Verifying Software Implementations on Hardware

In this section, we describe how we built Coco, a tool inspired by Rebecca [Blo+18], for the verification of masked software implementations directly on CPU netlists. More concretely, we show how the problem of verifying masked software implementations can be mapped to a hardware verification problem by treating software as a sequence of control signals that dictate the data/control flow within a CPU. This approach comes with the advantage that we can directly verify assembly implementations and observe a wider range of side-effects that could reduce the protection order of the tested software implementations. Previous works in this direction require modified software implementations and only consider a select amount of CPU side-effects that have been discovered in empirical evaluations [Bar+19; Bar+21].

First, we cover necessary background on masking and Rebecca. We then show that the classical probing model [ISW03] is not suitable for hardware/software co-verification and propose the so-called *time-constrained probing model* that can be seen as a stricter version of previously used models for software verification. We then discuss all improvements that we performed on top of Rebecca, such that hardware/software co-verification becomes feasible, ultimately leading to Coco. Coco's complete verification flow is described in Section 5.

[2] https://github.com/IAIK/coco-alma,
https://github.com/IAIK/coco-ibex

2.1. Background on Masking

Masking is a prominent algorithmic countermeasure against power analysis attacks [Cha+99]. In a nutshell, masking is a secret-sharing technique that splits intermediate values of a computation into $d+1$ uniformly random shares, such that observing up to d shares does not leak any information about the underlying value. The used masking scheme determines the number of masks d, and results in a dth-order masking scheme. In classical Boolean masking, the sharing of a native variable s, when split into $d+1$ random shares $s_0 \dots s_d$, must satisfy $s = s_0 \oplus \dots \oplus s_d$. Hereby, $s_0 \dots s_{d-1}$ is chosen uniformly at random while $s_d = s_0 \oplus \dots \oplus s_{d-1} \oplus s$. This ensures that each share s_i is uniformly distributed and statistically independent of s. For example, in a first-order masking scheme ($d = 1$), the secret variable s is split up into two shares s_0 and s_1, such that $s = s_0 \oplus s_1$. s_0 is chosen runiformly at random, while $s_1 = s \oplus s_0$.

When implementing masked cryptographic algorithms, dealing with linear functions is trivial as they can simply be computed on each share individually. However, implementing masking for non-linear functions requires computations on all shares of a native value, which is more challenging to implement in a secure and correct manner, and thus the main interest in literature [Bar+17; Bel+17; Cnu+16; GM17; GMK16; ISW03; Rep+15].

2.2. Background on REBECCA

Rebecca [Blo+18] is a tool for the formal verification of masked hardware implementations. Simply speaking, given the netlist of a masked hardware circuit, together with labels that indicate which input shares belong together, Rebecca can determine if the separation between shares is preserved throughout the circuit. More formally, Rebecca checks if a circuit is secure in the glitch-extended version of the original probing model by Ishai et al. [ISW03], which we refer to as the classical probing model. In general, the probing model defines the attacker's abilities in terms of the number of used probing needles, which are placed on a wire in a circuit and allow to observe the respective value from the wire. In the classical probing model, an attacker can place up to d probing needles in a circuit, which allows the observation of up to d intermediate values throughout the computation. A circuit is said to be dth-order protected if an attacker who combines the recorded information cannot infer information about native values.

The Verification Flow of REBECCA Rebecca operates on the netlist of a pipelined masked hardware circuit. A masked hardware circuit consists of linear gates (XOR, XNOR), non-linear gates (AND, OR), registers and constants, that are all connected by wires. Inputs are gates with indegree zero, such as the clock signal or the input state of a cipher.

The circuit inputs are annotated with labels to express their purpose in the masking scheme, which can either be a *share*, a *mask*, or *public*. A *share* represents

a share of a secret value, a *mask* is a fresh uniformly-distributed random value, and *public* means that it is not important for the masked implementation. These labels are propagated through all gates of the circuit, following a list of propagation rules. The circuit is not secure in the classical probing model if there is a gate that correlates with a native secret, i.e., allows an attacker probing the gate to deduce information about the native secret.

Rebecca is able to prove the glitch-resistance of masked hardware circuits. Glitches may arise in the combinatorial logic, and are caused by various physical hardware properties, including different wire lengths. Rebecca takes glitches into account by modeling the *stable* and *transient* correlation of gates. Stable correlations refer to the final values of the signals, whereas transient correlations refer to all intermediate signal values before the circuit stabilizes.

Fourier Expansions and Leakage Checks In order to check for correlation, Rebecca uses *correlation sets*. A correlation set is bound to a specific gate in the circuit and describes which information an attacker can learn by placing a probe on the gate. These sets are derived from the Fourier expansion of Boolean functions [ODo14]. Fourier expansions represent Boolean functions as a polynomial over the real domain $\{1, -1\}$. Examples of Fourier expansions are shown in Appendix A.

A function correlates to a linear combination of its inputs if the correlation term representing the linear combination has a non-zero correlation coefficient. Rebecca applies a very conservative over-approximation of these coefficients and derives correlation sets from these. Correlation sets contain terms with non-zero correlation coefficients while omitting the exact value of the coefficients. A first-order leakage test for a secret s checks whether a correlation set of any gate contains a term where all shares of s are present without being masked by a random value (a mask or an incomplete sharing of another secret). Explicitly constructing the correlation sets and performing these checks is infeasible, which is why Rebecca encodes everything as a pseudo-Boolean formula and checks for satisfiability with the SMT solver Z3 [MB08].

2.3. Probing Models for Software Verification

The complexity of a power analysis attack is determined by the number of intermediate values that an attacker needs to learn from a power trace by placing probing needles (probes) in a circuit. The number of probes corresponds to the order of an attack and the attack complexity grows exponentially with the order [Cha+99]. The classical probing model for hardware allows an attacker to observe all values and transitions at a chosen location within a hardware circuit, and therefore does not express this increase of complexity, but corresponds to a much more powerful attacker. For example, consider the case where an attacker is probing the write port of a CPU register file. Then, an attacker will always observe all intermediate values and can break masking schemes with arbitrary

protection order. Consequently, authors have fallen back to more restrictive probing models for the verification of masked software implementations.

Tools like maskVerif or *Tornado* are based on a probing model in which a dth-order attacker on software implementations can observe up to d intermediate values of the computation (+ transition effects). However, this implicitly excludes the attacker from observing more than two intermediate values at one probing location, even though CPU registers very likely contain multiple intermediate values throughout the software execution. Even though the essence of higher-order attacks is captured, it fails to represent that observing combinations of more than two intermediates is possible in practice.

Time-Constrained Probing Model We introduce the Time-Constrained Probing Model to model the capabilities of an attacker who performs power analysis attacks of a given order. The time-constrained probing model constrains the classical probing model such that the complexity of higher-order attacks is represented. In addition, it captures hardware effects and leads to situations where an attacker can observe more than two intermediate values at one probing location. Hardware effects, like glitches, occur frequently in practice and have been shown to be exploitable in the context of masked implementations [Fau+18; Moo+19; NRS11].

In the time-constrained probing model, an attacker possesses d probes. Each probe can be used to measure information in one specific clock cycle and at one specific location. The attacker can distribute the d probes spatially and temporally. Hence, the attacker can perform d measurements at different locations in the same clock cycle, or probes at the same location in different clock cycles, or a mix of both. A masked software implementation is dth-order secure in the time-constrained probing model if an attacker cannot combine the recorded information to learn anything about native values.

2.4. Co-Verification Methodology

While Rebecca is limited to the verification of pipelined masked hardware circuits, Coco aims at the co-verification of software and hardware, i.e., verifying the execution of masked software implementations directly on a processor's netlist. Consequently, Coco requires some knowledge about how concrete programs influence the data/control flow within the CPU. We then need to extend Rebecca such that the verification method is aware of the software execution.

In the following, we first briefly outline the workflow of Coco, broken into 4 steps. Steps 1-2 give intuition into how the execution of software can be combined with an otherwise purely hardware-focused verification method. Steps 3-4 then describe Coco's verification method. The remainder of this section describes Step 3 in more detail.

Step 1 We use Verilator [Sny22] to execute a masked assembly implementation

Table 1.: Definition of the stable (S_x^t) and transient (T_x^t) correlation sets of gate x in cycle t. We use the operator $\otimes$ as the element-wise multiplication of two correlation sets.

Gate type of x		Definition of S_x^t	Definition of T_x^t
Constant		$\{1\}$	$\{1\}$
Negation	$x = \neg a$	S_a^t	T_a^t
Register	$x \Leftarrow_R a$	S_a^{t-1}	$\widehat{S}_a^{t-1} \otimes \widehat{S}_a^t$
XOR	$x = a \oplus b$	$S_a^t \otimes S_b^t$	$\widehat{T}_a^t \otimes \widehat{T}_b^t$
XNOR	$x = \overline{a \oplus b}$		
AND	$x = a \wedge b$	$\widehat{S}_a^t \otimes \widehat{S}_b^t$	$\widehat{T}_a^t \otimes \widehat{T}_b^t$
OR	$x = a \vee b$		
Multiplexer	$x = c \ ? \ a : b$	$\widehat{S}_c^t \otimes (S_a^t \cup S_b^t)$	$\widehat{T}_c^t \otimes \widehat{T}_a^t \otimes \widehat{T}_b^t$

on a given CPU hardware design via a cycle-accurate simulation. From the simulation, we extract a so-called *execution trace* which contains concrete values for all CPU control signals in all execution cycles. We require implementations with a constant control flow using Boolean masking and therefore, these control signals are the same for all inputs to that software implementation.

Step 2 We annotate which registers or memory locations hold the shares of a native value at the start of the software execution. Additionally, we need to specify the masking order of the software implementation and the number of cycles that should be verified.

Step 3 We capture the correlations of each logic gate and register in the processor by constructing correlation sets throughout each clock cycle. For this purpose, we improve and extend the set of stable and transient propagation rules used by Rebecca. Most importantly, we reformulate them such that they can be made execution-aware. Knowing the exact values of control signals at each point during the execution allows Coco to simplify the correlation sets under certain circumstances. In turn, we obtain a tighter over-approximation and reduce erroneous leakage reports.

Step 4 We encode the resulting correlation sets as a propositional Boolean formula and use a SAT-solver to check for leakage. In case the implementation is insecure, the exact gate in the netlist and execution cycle is reported. Tracking correlation sets naively is infeasible since their size grows exponentially with the number of secret shares and masks. Our encoding includes the circuit structure, correlation propagation rules and security constraints. Although Rebecca already applies this approach, their SAT encoding is incompatible with our execution-aware propagation rules and not efficient enough for circuits as large as processors.

Execution-Aware Stable Correlation Sets In Coco, we apply an over-approximation of the Fourier expansions of Boolean functions by building execution-aware correlation sets S_x^t which track the non-zero correlation terms of gate x in cycle t. For reasons of simplicity, we also define the biased correlation set $\widehat{S}_x^t = \{1\} \cup S_x^t$. In Step 2 of the verification process, we decide on the initial correlation terms by providing labels for registers and memory locations. For example, if we label register x as the first share s_1 of the secret s, then its initial correlation set is $S_x^0 = \{s_1\}$. Correlation terms of consecutive gates are derived by propagating these labels through the whole circuit, using the definitions of stable correlation sets, until the initial registers are reached again. The register's labels are updated accordingly and the propagation restarts. This process is repeated for every cycle, until the execution finishes.

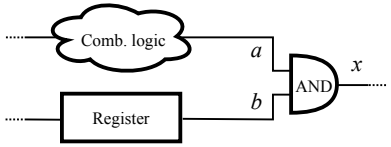
Table 1 shows the definitions of stable correlation sets S_x^t used by Coco. Constants only correlate to the constant term 1. Negations only change the sign of the coefficients in the Fourier expansion, so the correlation set stays the same. Registers inherit the stable correlation set their input had at the end of the last cycle. The stable correlation set of linear gates (XOR, XNOR) is computed as the element-wise multiplication ($\otimes$) of the correlation set of the gate inputs. Similarly, the definition for non-linear gates is calculated as the element-wise multiplication of the biased correlation set of the gate inputs.

Unlike Rebecca, our verification tool supports multiplexers. Therefore, in Equation 1, we propose the Fourier expansion of multiplexer gates.

$$\text{MUX} \quad F(c\ ?\ a : b) = \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ac + \frac{1}{2}bc \tag{1}$$

A detailed derivation of the coefficients is given in Appendix A2. Consequently, the correlation set for multiplexers combines the stable correlation sets of all inputs.

The resulting over-approximation S_x^t is sound but not always tight. This means that the stable correlation set contains at least all correlation terms with non-zero coefficients, but might also contain terms that have a zero coefficient. In other words, all real leaks are always detected, but sometimes leaks could falsely be reported. Unlike Rebecca, Coco tightens the over-approximation and circumvents the necessity to apply the full sets in some cases, which reduces the amount of false positives. The propagation rules for gates which have at least one *public* input can, depending on the concrete value of the input, be simplified by substituting correlation sets with constants. The concrete values can be obtained from the execution trace. For example, if there exists a mulitplexer $c\ ?\ a : b$ and we know that c is *public* and has the concrete value FALSE, the result of the multiplexer will only correlate to terms in S_b^t.



	Cycle n	Cycle $n+1$	Cycle $n+2$
Boolean values according to trace			
b	1	0	0
x	a	0	0
Stable correlation sets			
S_x^t	S_a^n	$\{1\}$	$\{1\}$
Transient correlation sets			
T_x^t	T_a^n	T_a^{n+1}	$\{1\}$

Figure 1.: Example of simplifications made to the propagation rule of an AND gate in three consecutive cycles, exploiting execution-awareness.

Execution-Aware Transient Correlation Sets Hardware effects like transitions and glitches cause information leaks, which cannot be captured by stable correlation sets. Therefore, we introduce transient correlation sets T_x^t for a gate x in cycle t and the biased representation $\widehat{T}_x^t = \{1\} \cup T_x^t$. T_x^t contains at least all the correlations an attacker can observe throughout the duration of one cycle. Additionally, it contains spurious terms that make efficient calculations easier while still yielding an over-approximation, albeit a less tight one.

The definitions of transient correlation sets T_x^t are shown in Table 1. For constants and negations, the definition of the correlation sets is identical to the stable case. An attacker probing a register can learn the current stable value, the old stable value, and their linear combination due to transition leakage. Therefore, probing a register does not reveal any transient information, as registers synchronize the circuit and do not change throughout a clock cycle. Non-linear and linear gates leak the same amount of information in the transient case. Glitches can cause a linear gate to forward either of its inputs because they do not necessarily update simultaneously. Similarly, due to the transition from the previous stable signal value to the current transient signal value, an attacker can observe both, as well as their linear combination. The over-approximation in Table 1 does not state this directly. Instead, this is implied by the transient correlation sets for registers, which make sure that an attacker probing any gate also sees the old stable value of that gate. Therefore, as $S_a^{t-1} \subseteq T_a^t$, gates using a as an input observe both old and new signal values of a. In the transient case, Coco treats multiplexers similarly to linear and non-linear gates. Our over-approximation just assumes that a multiplexer leaks all possible linear combinations of the transient values of all of its inputs.

Just like stable correlation sets, transient correlation sets are also affected by

concrete signal values obtained from the execution trace. However, glitches make simplifications due to execution awareness harder and less effective. They are still possible, as long as we keep track whether a given signal can cause a glitch or not. We use a method similar to what was proposed by Thompson et al. [TM04] to track the stability of a given signal. This method is summarized by the following rules:

- Registers that have not changed their value during a transition from cycle $t-1$ to cycle t cannot produce glitches, as their signals are inherently stable.
- If all inputs of a logic gate are stable, the output of the logic gate cannot cause glitches either.
- Non-linear gates and multiplexers can still produce stable signals, even if one of its inputs is unstable. This depends on the gate's physical properties, which can prevent glitches, e.g. AND gates with one unstable and one stable FALSE input, OR gates with one unstable and one stable TRUE input.

The gate stability propagates through the circuit for any given clock cycle, starting at registers and continuing until the stability of all gates is determined. After computing which circuit gates produce stable signals, we use this to apply simplifications to transient correlation sets using the same method as for stable correlation sets.

Example of Execution-Aware Simplifications Consider an AND gate $x = a \wedge b$, where b is the output of a register and a is calculated by some combinatorial logic, as shown in Figure 1. For simplicity, assume that the value of b is *public*, and that the value of a, as well as the stable and transient correlation sets, do not change throughout cycles n to $n+2$, i.e., $S_a^n = S_a^{n+1} = S_a^{n+2}$ and $T_a^n = T_a^{n+1} = T_a^{n+2}$.

From the execution trace we know that $b = 1$ in cycle n and $b = 0$ in cycles $n+1$ and $n+2$. Knowing b allows us to apply the simplifications $S_x^n = S_a^n$ and $S_x^{n+1} = S_x^{n+2} = \{1\}$. Now consider the same circuit when glitches are present, and assume that $b = 1$ was a stable signal in cycle n. In cycle $n+1$, it is possible that the signal from a arrives at x before the new value $b = 0$. Therefore, the simplifications due to execution awareness cannot be applied and, $T_x^{n+1} = T_x^n = T_a^n$. However, in cycle $n+2$, we can apply the simplification because the value of b is stable and, thus, $T_x^{n+2} = \{1\}$.

3. Problems and Fixes in the IBEX Core

In this section, we first describe the RISC-V IBEX core, our target processor. We analyze the RISC-V IBEX core using COCO to identify implementation details that prevent the leakage-free execution of masked software implementations.

Afterwards, we propose corresponding fixes, either directly in hardware, or as a constraint for masked software implementations. The outcome of our analysis is a *secured* hardware design of the Ibex core. We discuss secure options for data memory in Section 4 and then verify the entire design in Section 5.

When executing a masked software implementation on Ibex, secret shares are initially stored in the register file and the data memory. The instructions of the program work on the shares by changing them and moving them through the CPU and the memory system. All these actions cause potential leakage. In order to analyze and detect these leakage sources, we work with a comprehensive set of masked software implementations that includes (higher-order) masked And-gates, a second-order masked Keccak S-box, and a first-order masked AES S-box implementation. All test programs are written in RISC-V assembly and then executed on the Ibex core, producing a cycle-accurate execution trace. The execution trace in combination with the exact storage location of the secret shares (registers or memory locations) is then processed by Coco, which automatically runs the verification and reports leakage sources by specifying the exact cycle and gate in the netlist. We then manually inspect the gate in the netlist, introduce the corresponding hardware fixes and re-evaluate the design until no leaks were dectected anymore.

Our analysis has revealed several leakages caused by the Ibex core. First, Coco has confirmed the typical problems of masked software implementations that have already been identified by previous works, such as overwriting or successively accessing shares that correspond to the same native variable [Bal+14; Bar+15; PV17; She+21]. While fixing such problems in hardware would, in principle, be possible, it would be very costly. We decided to accept these leakages and instead write all our masked implementations in a way such that they fulfill the following two constraints:

$\texttt{C1}_{\texttt{CORE}}$ Shares of the same secret must not be accessed within two successive instructions.

$\texttt{C2}_{\texttt{CORE}}$ A register or memory location which contains one share must not be overwritten with its counterpart.

However, although these design principles prevent known leakage sources, Coco has revealed many more leakages. In particular, it identified leakages in the register file, the computational units (ALU, MD, and CSR) as well as in the LSU. We now discuss all of these identified problems for the different components of the CPU and present corresponding solutions in hardware to prevent these leakages.

3.1. Targeted Processor Platform

The Ibex core[3] is a free and publicly available 32-bit CPU design that features a two stage in-order single-issue pipeline that is divided into Instruction Fetch (IF) and Instruction Decode/Execute (ID+EX). Its performance is roughly comparable to the ARM Cortex-M0. The main components of Ibex are the register file, the Arithmetic Logic Unit (ALU), the Load-Store Unit (LSU), a unit for multiplications and divisions (MD), the Control and Status Register (CSR) block, and several functional units for processor control, including the decoder and controller.

For our analayis we use Ibex core commit `863fb56eb166d`. We configure Ibex to use the RV32I instruction set and the C (compressed instructions), M (multiplication/division) and Zicsr (control and status register) extensions. Other features like physical memory protection and the instruction cache are disabled.

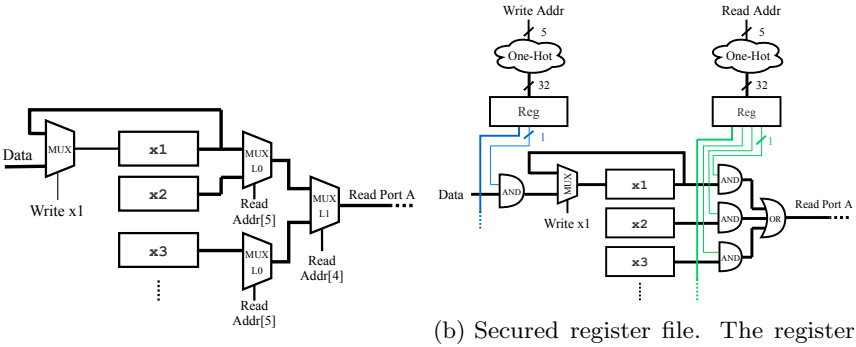
We select Ibex as the target core because it has a relatively simple microarchitecture, which makes it easy to demonstrate Coco and explain the hardware fixes. Although the core complexity is rather low, it still contains the most important components which are part of every modern processor, for example the register file. Additionally, the Ibex core has gained a lot of attention recently as beging part of the PULP Platform [ETH] and the OpenTitan project [low19].

However, we want to stress that Coco can be used to analyze any other processor, as long as the netlist is available in either Verilog or System Verilog and the masked software implementations have a constant control flow. This includes also larger RISC-V cores, for example the 32-bit CV32E40P (formerly RI5CY) [Opea] and the 64-bit CVA6 (formerly Ariane) [Opeb], but also other non-RISC-V processors, for example the ARM Cortex-M4. Note that the netlist does not necessarily have to be open source. For example, users in industry to which the netlist of the ARM Cortex-M4 was disclosed, could use Coco to perform verification of ARM-based masked assembly implementations. Additionally, the problems found in the Ibex core are conceptually the same in larger cores, since the basic building blocks are the same. Therefore, the proposed solutions can also be easily mapped to larger cores.

3.2. Register File

The register file of the Ibex core consists of 32 32-bit registers, labeled `x0-x31`, where `x0` is hard-wired to the value 0. Although there exist multiple options of how concrete register files could be constructed, on a conceptional level, the design will be similar to the sketch shown in Figure 2a. There are two read ports (A and B), and a write port, that are controlled by 5-bit address signals. The 32 registers are connected to a multiplexer tree of depth five, whose selection signals are the respective bit of the read address. If an instruction writes a value to a register, the 32-bit write data either originates from the ALU, the CSR Unit,

[3] https://github.com/lowRISC/ibex



(a) Original register file. A multiplexer tree is used to read registers based on the 5-bit read address. Writing is done via a multiplexer, controlled by a 1-bit write-enable signal, which is derived from the write address.

(b) Secured register file. The register output is additionally gated and the multiplexer tree is replaced by a tree of OR gates. The writing mechanism remains unchanged, except that it is extended by an additional AND gate for the write data.

Figure 2.: Original and secured register file of the IBEX core.

or the LSU. A multiplexer before each register controls if the register content is updated, depending on the write-enable signal, which is derived from the address.

Problem: Switching Wires in the Multiplexer Tree The transition from one secret share to another may be observable on a wire connecting two levels of the multiplexer tree. This happens primarily whenever two secret shares are read in consecutive cycles, but also when accessing registers unrelated to secret shares. For instance, assuming that the secret shares are in registers `x1` and `x2`, reading register `x3` in the first cycle and `x4` in the second cycle causes the fifth bit of the read address to switch from one to zero. An attacker observes leakage on the output wire of the first L0 multiplexer, which switches from `x1` to `x2`.

Problem: Glitchy Address Signals The read and write address signals are not guaranteed to be glitch-free since they come out of combinatorial logic. We identify the transitions of the wires in the multiplexer tree as a source of leakage because it can switch from the value of a secret share in the register to the data written to *any* other register. Additionally, transitions from one secret share to another can be observed on the output of the multiplexers before a register.

Problem: Unintended Reads The IBEX core reads data from the register file in *every* instruction, even in cases were the current instruction does not require any operands. For example, `lw x1, 5(x20)` will result in a read to registers `x20`

and `x5` because bits 15-19 and 20-26 of an instruction are always interpreted as operand addresses.

Solution: Register Gating All three described problems are difficult to address in software since their effects often depend on the concrete hardware layout. A pure software solution could eliminate the problem of unintended reads, but becomes more complex as the length of a program grows and is completely unfeasible for larger implementations. Software mitigations are insufficient to solve the problem of glitchy address signals and transition leakage in the multiplexer tree. Therefore, we fix this problem in hardware using a gating mechanism for each register, as shown in Figure 2b. After each register, we place an AND gate, that takes the register value as the first input operand. The second operand of this AND gate is the register read address, encoded into a 32-bit one-hot signal, where each bit represents the gate value for a single register. Consequently, the whole multiplexer tree can be replaced by a simpler tree of OR gates. From a verification aspect, we discuss this solution in Figure 1. In this concrete example, the one-hot encoded enable signal is stored in the register while the combinatorial logic represents the CPU register. Since at most one bit is set in the one-hot signal, at most one register gate is opened, and either the correct register value or zero can be read from the register file. This gating mechanism prevents the problem of switching wires in the multiplexer tree, and unintended reads because we only enable gating when the instruction requires a read. We prevent glitches on the one-hot signal by computing it in the IF stage, and storing it in an intermediate register so that it is guaranteed to be stable when it reaches the ID+EX stage. We apply the gating mechanism to both read ports. Likewise, register writes are also gated with a separate pre-computed value in a one-hot register by placing an AND gate before the write multiplexer.

3.3. Computation Units

Computation units such as the ALU, MD, and CSR are directly connected to read ports of the register file. The results produced by them go directly into a multiplexer, selecting the intended computation result for the register write port. In other words, the IBEX computation units are always active, even when they are not required by the current instruction.

Problem: Always-Active Computation Units Assume the b-bit secret $\boldsymbol{s}$ is shared into two shares $\boldsymbol{s}_0 = (s_{0,1}, ...s_{0,b})$ and $\boldsymbol{s}_1 = (s_{1,1}, ...s_{1,b})$, such that $\boldsymbol{s} = \boldsymbol{s}_0 \oplus \boldsymbol{s}_1$. Traditionally, $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ are both stored in one register each, but there are other ways the bits of shares can be stored. For example, in 2017, Barthe et al. [Bar+17] proposed parallel implementations of higher-order masking schemes, where $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ are distributed over b registers $\boldsymbol{r}_1, ...\boldsymbol{r}_b$. In their scheme, the first bit of $\boldsymbol{r}_1$ stores $s_{0,1}$, while the second bit stores $s_{1,1}$.

The standard IBEX core does not allow leakage-free implementations of such masking schemes since parts of ALU, MD, and CSR units are always active and combine the bits of each read port signal. More concretely, when using a parallelized masking scheme, the execution of a simple bit-wise `and` instruction leaks since, e.g., the adder unit combines the bits from the first input operand, and thus might leak $s_{0,1} \oplus s_{1,1}$.

Solution: Computation Unit Gating The problem of always-active computation units is very hard to mitigate in software. Therefore, we use a gating mechanism in hardware similar to the one in the register file. More concretely, we use additional AND gates at the inputs of each computation that are connected to respective enable bits, which are precomputed in the IF stage and depend on the next instruction. This also has the other positive side-effect that the reduced circuit activity results in an overall lower power consumption of the CPU, reducing the overall switching activity in the circuit.

3.4. Load/Store Operations

The LSU implements a state machine that is responsible for communicating with the external memory. The state machine mainly handles the correct interaction with data/instruction memory including misaligned memory accesses.

Problem: Hidden LSU State Accessing 32-bit words at addresses that are not 32-bit aligned always results in two consecutive fetch operations of the corresponding memory words. An internal register is then used to buffer the first memory word until the second memory word is available. This internal buffer is only updated once a misaligned memory access occurs. Programs can, therefore, cause unintended leaks by loading a share into the LSU buffer. The value in this buffer will then potentially be combined with all values that traverse the LSU from this time on.

Solution: Clear Hidden LSU State We can avoid this leakage source in software by performing a misaligned memory access to a non-secret value, which clears the LSU buffer. However, we solve this problem in hardware since it does not produce any additional overhead, and no additional software design constraints are necessary. A memory access executed by the IBEX core requires at least two clock cycles. In the last cycle, the read memory word is given back to the LSU. In fact, clearing the hidden LSU buffer in the first cycle, i.e., at the beginning of a memory access, eliminates this leakage source.

3.5. Hardware Overhead

In order to analyze the additional hardware overhead of the security fixes implemented in our design, we compare the chip area in kGE as well as the maximum

Table 2.: Area consumption of the Ibex core in kGE. The area consumption of the whole design (Total) and parts (register file, IF stage) are reported. The area consumption of the ID+EX stage is omitted because there is no overhead. The total area overhead of the design with all security fixes enabled is around 10%.

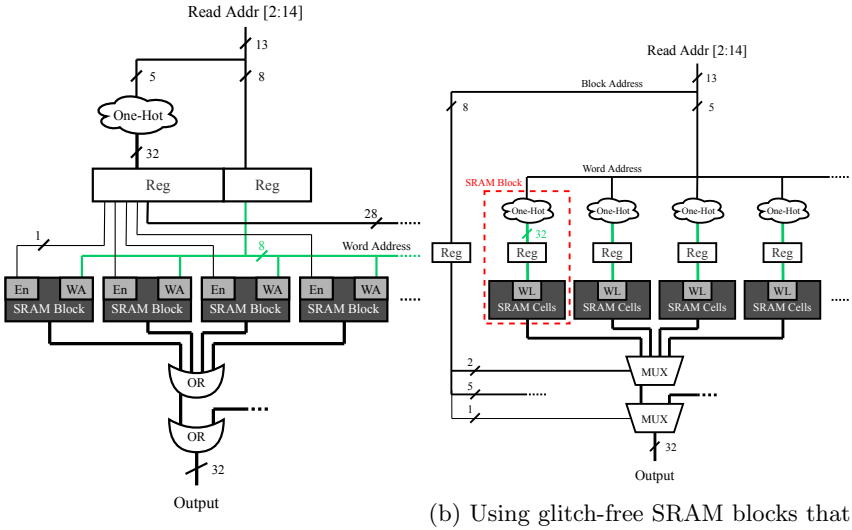
Design	Total		Register File		IF stage	
	Total	Overhead	Total	Overhead	Total	Overhead
#1 Base design	20.2	-	9.8	-	3.0	-
#2 BD + secure register read	20.5	1.5 %	9.4	−4.1 %	3.6	29 %
#3 BD + secure register write	21.9	8.4 %	11.0	12.2 %	3.4	13 %
#4 BD + secure register read-/write	22.1	9.4 %	10.7	9.1 %	4.0	33.3 %
#5 BD + disabled MD/ALU/CSR unit	20.4	0.9 %	9.8	0 %	3.1	3.3 %
#6 Secured design	22.2	9.9 %	10.7	9.1 %	4.0	33.3 %

operating frequency of the Ibex base design with our secured design. We use Cadence Genus Synthesis Solution 19.11-s087_1 for synthesis. The used technology is f130LL.

We disable the `ungroup_ok` option for all modules in the core, which preserves the hierarchy of the design. This allows us to investigate the area consumption of every submodule on its own, although it might prevent certain optimizations. We can also exclude the area consumed by SRAM and the instruction ROM from the analysis since they do not belong to the Ibex core.

Table 2 shows the area consumption of the Ibex core in different configurations. The unmodified Ibex core (design #1) requires in total 20.2 kGE. Enabling secure register reads by gating (design #2) increases the total chip area by 1.5 %. This is mainly due to the additional two 32-bit registers required in the IF stage. The size of the register file even decreases, because Or gates replace the multiplexer tree. However, register writes introduce more area overhead due to the additional And gates. In design #5, main overhead comes from the four 1-bit gating-registers in the IF stage and the And gates used for gating in the total core overhead. In summary, all our security fixes increase the total area of the Ibex core by 9.9 %.

We do not expect a major latency overhead of our modifications. In the core, we mainly shifted the address decoding from ID to IF stage, which might slightly increase the latency of the IF stage. The same holds for the ID stage, where the multiplexer tree is replaced by a tree of Or gates and a layer of additional And gates. The computation unit gating and clearing the hidden LSU state will also affect latency in the ID stage. Latency considerations according to the SRAM are discussed in Section 4. However, we keep a detailed investigation as an open research question for the future.



(a) Using glitchy SRAM blocks. The stable one-hot encoding of the higher address bits is computed outside of the SRAM blocks.

(b) Using glitch-free SRAM blocks that compute a stable one-hot encoding of the lower address bits. The word line (WL) selects the active word (see also Figure 4).

Figure 3.: Two options of adding SRAM to our Ibex core.

4. Problems and Fixes in Data Memory

In this section, we discuss how data memory, more specifically SRAM, can be integrated into our secured Ibex core so we can formally prove the leakage-free execution of masked software implementations for the entire system. Typically, microprocessors such as ARM Cortex-M devices feature a Harvard architecture, which means that dedicated memory modules are used for data/instruction memory (based on SRAM/Flash technology). Especially on low-end devices, without sophisticated branch prediction and cache architectures, this design choice improves overall performance since simultaneous memory accesses to both memory modules are possible. For our purposes, dealing with instruction memory is comparably easy since instructions only dictate the data/control flow. They are not directly involved in any computations and are thus not labeled as shares in our verification. Hence, from a hardware perspective, we do not need to take any special precautions when adding instruction memory to our Ibex core.

The situation becomes more complicated for data memory, as it plays an important role for masked software implementations that cannot hold all intermediate values of a computation in its register file. At first glance, one could consider applying the same design strategy, as used for the register file (cf. Figure 2b),

also to the data memory. However, one-hot encoding does not scale well with larger address spaces and would result in impractical hardware overhead.

Consequently, we need to discuss options that keep the hardware overhead reasonable while still allowing correctness proofs for the entire CPU design. In the following, we discuss two such options that utilize partially one-hot encoded address signals and result in different trade-offs between hardware overhead and the number of rules that need to be followed by masked software implementations. The first option utilizes one-hot encoding in the upper address bits, i.e., for selecting SRAM blocks, and does not make any assumptions on the inner workings of the SRAM blocks. The second option describes how one-hot encoding in the lower address bits can be used to build "glitch-free" SRAM blocks that can then easily be added to our Ibex core without any hardware overhead.

4.1. MSB One-hot Address Encoding

The first viable option of using partial one-hot encoding for data memory involves using one-hot encoding for the higher bits of the address signal, as illustrated in Figure 3a. In this example, we consider the case of a low-end 32-bit device with 32KB of RAM that can be addressed on word granularity with 13-bit address signals (i.e., using bits 2 to 14 from the original 32-bit signal). First, we extract 13 bits from the original 32-bit address signal. This 13-bit signal is then further split up into a 5-bit block address (later expanded to a 32-bit one-hot signal) and an 8-bit word address for selecting a word within one SRAM block. This design choice ensures that no glitches can occur across SRAM blocks, yet they could still occur between the words of a single SRAM block. More concretely, when considering a masked software implementation that operates on a secret s, represented by the shares $s = s_1 \oplus s_2$, then our construction results in the following software constraints for SRAM usage:

$\texttt{C1}_{\texttt{SRAM}}$ Storing both, s_1 and s_2, in separate SRAM blocks is fine as long as they are not accessed in immediate succession.

$\texttt{C2}_{\texttt{SRAM}}$ Storing s_1 and s_2 within the same SRAM block can result in potential leaks and thus needs to be avoided.

The hardware overhead of utilizing one-hot encoding in the higher address bits is mainly determined by the additionally needed one-hot encoder circuitry and one 40-bit register. On the other side, when comparing Figure 3a to Figure 3b, one can also see that the Mux-tree, used for selecting the SRAM output, can be replaced by a simpler, and thus cheaper Or-tree. Overall, and when compared to the typical area of SRAM blocks, we do not expect any noticeable hardware overhead of this construction. From a latency perspective, there is no delay as long as the one-hot encoding can be performed in the cycle before the actual lookup. We expect this to hold for most designs.

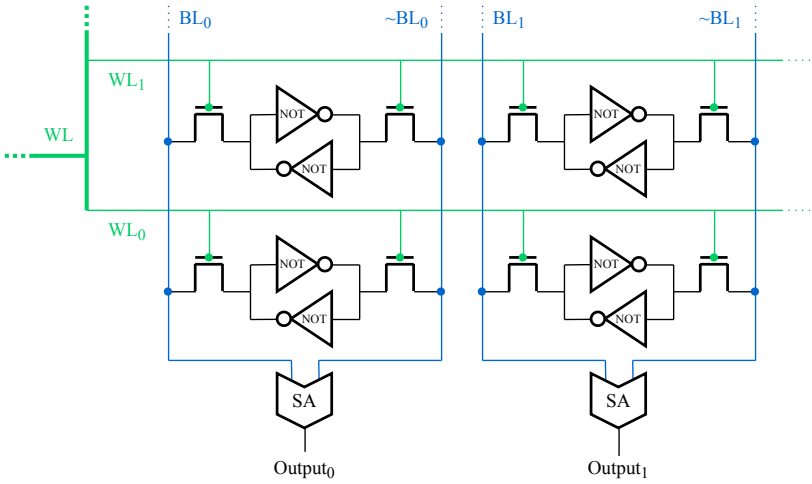


Figure 4.: Typical layout of SRAM cells. Each pair of NOT gates represents a 1-bit memory cell. The one-hot encoded word line (WL) selects the active word. The bit line BL_i connects bits at location i from all words. The negated BL signal, together with the differential sense amplifier (SA), help achieving stable output values faster.

4.2. LSB One-hot Address Encoding

Another option of utilizing partially one-hot encoded address signals consists of using one-hot encoding only for certain less significant bits of the address signal, as illustrated in Figure 3b. In this case, the 13-bit address signal is divided into an 8-bit block address (for specifying the SRAM block) and a 5-bit word address that is later expanded to a 32-bit one-hot signal (for specifying a word within an SRAM block). This construction will, similarly to the register file, as discussed previously (cf. Section 3.2), eliminate glitches between words of the same SRAM block, except for the case when they are accessed in immediate succession. Consequently, when operating with the shares s_1 and s_2, masked software implementations need to follow the following constraints:

$\texttt{C1}_{\texttt{SRAM}}$ Storing both, s_1 and s_2, within the same SRAM block is fine as long as they are not used in immediate succession.

$\texttt{C2}_{\texttt{SRAM}}$ Storing s_1 and s_2 in different SRAM blocks can result in potential leaks and thus needs to be avoided.

When looking at the standard design of SRAM cells in Figure 4, one can observe that the word line (WL) needs to be a one-hot encoded signal while each bit line (BL) is connected to one bit location of all words within one SRAM block,

thereby essentially functioning as an OR gate. On a conceptional level, this is similar to the construction in Figure 3b, were we use additional registers to ensure a stable WL signal.

In other words, if a given SRAM block has a layout that already achieves internally stable WL signals in practice then no hardware modifications are required and an ordinary MUX-based output selector can be used. Of course, it is generally not easy to tell if, or to what extent, an off-the-shelf SRAM block fulfills this requirement since they are full custom and partially analog blocks. In a typical SRAM row decoder design, an individual WL signal is derived by a single, wide NOR gate with a fan-in that is equal to the number of bits in the word address (see Section 2.7 in [PS08]). Roughly said, if the address signal is stable, then the low combinatorial depth of the row decoder likely only causes small glitches that could then be compensated with the custom circuit layout. Besides that, stable WL signals are also desirable from a power and latency perspective since (1) each WL signal can drive up to 64 transistors, glitches can hence significantly impact the power profile, and (2) the time until the differential sense amplifier (SA) output is stable strongly depends on the presence of glitches on the WL signals, which in return reduces the maximum operating frequency.

5. Co-Verification with Coco

In this section, we discuss the details of the workflow of COCO, our verification tool, and report the runtime effort for each step. We evaluate COCO using several benchmarks, including first-order and higher-order masked implementations executed by the secured IBEX processor and show that COCO can efficiently verify those. We run all our evaluations using a 64-bit Linux Operating System on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. Additionally, we practically evaluate our design using a first-order t-test on a SAKURA-G FPGA evaluation board.

5.1. Verification Flow

The verification flow implemented by COCO consists of four steps, as illustrated in Appendix B. The four steps are divided into three preprocessing steps (1)-(3), and the final verification step (4). The preprocessing steps are needed to join the masked assembly implementation of the cipher with the IBEX System Verilog sources into one single VCD execution trace, which is then used during verification. For all our experiments, we use the secured IBEX processor, which consists of the secured core and memory, as described in Sections 3 and 4. In detail, the verification flow is as follows:

(1) The masked implementation of the target cipher is compiled using the 32-bit RISC-V assembler. The resulting binary file is then converted into a Verilator [Sny22] testbench.

(2) We use Yosys [Wol16] to parse the hardware model, a set of System Verilog files, of the secured Ibex processor. Yosys (Yosys Open SYnthesis Suite) is an open-source framework which synthesizes and optimizes the model and produces a netlist of the circuit in Verilog format and as a graph, with gates as nodes and wires as edges.

(3) We run Verilator using the testbench created in (1) and the circuit netlist created in (2). It produces an execution trace of the masked cipher executed by the secured Ibex processor in VCD format.

(4) In the last step, the actual verification is done using a Python script. The script's input are the circuit graph, the VCD execution trace and the verification configuration. The verification configuration consists of the register label file, which specifies which registers or memory locations contain shares of a secret and which contain fresh randomness, the verification mode (stable or transient), the number of cycles which should be verified and the order of the masked cipher. Finally, the verification process outputs whether the execution is leakage-free or not, together with the cycle and gate number in which the leakage occurred.

Since the System Verilog support of Yosys is limited, we use the Symbiotic EDA Edition of Yosys (0.8+472), which works with a frontend of Verific in order to support System Verilog. Verilator 4.010 is used to create the execution traces. A Python script is used to create the SAT formulas, which are later solved by CaDiCaL 1.0.3.

In our experiments, we cannot work with real SRAM blocks for data RAM. Usually, one would use pre-build and pre-configured SRAM modules and instantiate them with a macro in the Verilog code. However, in that case, we can neither trace the behavior of the block during execution nor label memory cells. Therefore, we create a Verilog hardware model according to the LSB one-hot address encoding scheme, as described in Figure 3b, which behaves like a real SRAM module. The module is divided into 16 blocks consisting of 8 32-bit words each. Furthermore, we configure Ibex core to use 1 kilobyte of instruction memory for all test cases except the DOM AES S-box, where we use 4 kilobytes.

5.2. Evaluation of Preprocessing Steps (1) - (3)

Coco's preprocessing steps aim at preparing all resources for the verification. The runtime of the testbench creation (1) takes about 0.04 s for all our experiments. The runtime of the tracing part (3) is determined by the circuit size and number of cycles it needs to execute the masked software implementation with Ibex and takes 0.003 s per cycle. The parsing step (2) has to be run only once for the whole secured Ibex and takes about 7 min and depends mostly on the circuit size, including the size of instruction and data memory.

Table 3.: Verification of masked software implementations on secured Ibex using Coco. ✖ indicates intentionally broken implementations. Testcases with *reg.* omit memory accesses and perform all computations using registers. Runtimes stem from single-threaded executions on an Intel Core i7 notebook CPU with 16 GB of RAM.

Name	Runtime (cycles)	Leaking Cycle	Input Shares	Fresh Randomness	Verif. Stable	Runtime Transient
First-order						
DOM And *reg.* [GMK16]	13	-	4 × 32 bit	32 bit	3 s	11 s
DOM And *reg.*✖	13	12	4 × 32 bit	32 bit	2 s	12 s
DOM And [GMK16]	39	-	4 × 32 bit	32 bit	9 s	32 s
ISW And *reg.* [ISW03]	13	-	4 × 32 bit	32 bit	5 s	13 s
TI And *reg.* [NRR06]	17	-	6 × 32 bit	-	5 s	17 s
Trichina And *reg.* [Tri03]	19	-	4 × 32 bit	32 bit	5 s	19 s
DOM Keccak S-box *reg.* [GSM17]	89	-	10 × 32 bit	5 × 32 bit	25 s	2.6 m
DOM Keccak S-box *reg.*✖	88	70	10 × 32 bit	5 × 32 bit	20 s	2 m
DOM Keccak S-box [GSM17]	219	-	10 × 32 bit	5 × 32 bit	1 m	3.9 m
DOM AES S-box [BP12]	1900	-	16 × 16 bit	34 × 16 bit	18 m	4.75 h
Second-order						
DOM And *reg.* [GMK16]	34	-	6 × 32 bit	3 × 32 bit	9 s	43 s
DOM Keccak S-box [GSM17]	474	-	15 × 32 bit	15 × 32 bit	3 m	1.3 h
Third-order						
DOM And *reg.* [GMK16]	65	-	8 × 32 bit	6 × 32 bit	44 s	2.5 m

The result of (2) is a netlist of the secured Ibex processor in graph representation. The Ibex core, excluding data and instruction memory, consists of almost 27 000 gates. It is important to note that our hardware design is orders of magnitudes larger than designs considered by other verification tools. For example, Rebecca [Blo+18] performs verification on hardware circuits consisting of at most 200 registers and 3 000 non-linear gates, while maskVerif [Bar+19] and Silver [KSM20] consider circuits with up to 300 and 1 000 probing positions.

5.3. Evaluation of the Verification Step (4)

The verification results of the masked software implementation run on the secured Ibex processor, and their verification runtime are shown in Table 3. The table states the testcase in RISC-V assembly and how many cycles the execution takes. We report the number of labels provided by the user, divided into shares and fresh randomness. It is very important to note that each of these shares or random values is either 32 bit or 16 bit wide. Other verification methods often argue that a hardware circuit computing a masked cipher treats each bit in the same way, so it is sufficient to view a 32-bit register as one single share. However, in the Ibex processor, this is not the case, since logic in different computation units tends to treat each register bit differently. Therefore, we must label and check all 32 bits individually.

The selection of masked circuits covers different masked $GF(2)$ multipliers (And gates), including the Domain-Oriented Masking (DOM) And, Ishai-Sahai-Wagner (ISW) And, Threshold Implementation (TI) And and Trichina And,

but also larger implementations like the Keccak S-box and the AES S-box. Furthermore, we show that it is feasible to verify second-order and third-order implementations. Our benchmarks focus on the verification of non-linear parts of cipher implementations, similar to Rebecca, maskVerif and Silver, although the linear parts could easily be added to the implementation. Coco verifies all tested first-order masked multipliers in transient mode in less than 20 s. Larger testcases, for example, the DOM AES S-box, can be verified in a few hours.

In addition, we want to point out that errors in implementations can be found efficiently. Implementations marked with ✖ refer to implementations which cause side-channel leakage when executed with the secured Ibex because (1) masking is either done incorrectly on the algorithmic level, or (2) masking is correct on the algorithmic level but software constraints are not satisfied. DOM And *reg.*✖ is a first-order DOM multiplier based on [GMK16], in which fresh randomness is added to the shares too late. The stable verification reports an error in cycle 12 in a gate belonging to the ALU. DOM Keccak S-box *reg.*✖, based on [GSM17], does not follow constraint $\texttt{C2}_{\text{CORE}}$. This flaw is reported by transient verification in cycle 70 and appears directly on the read port of the register file. The verification runtime of an insecure implementation is similar to that of a secure implementation because the verification terminates as soon as the leakage check for any share fails.

The total verification runtime can be split into the construction and solving of the SAT formula. In our experiments, solving the SAT formula requires considerably less time than constructing the SAT formula, which is linear in the number of gates in the netlist, i.e., the number of registers and the size of the combinatorial logic between these registers. Hence, for moderate increases of the problem size, for example through larger cores having multiple ALUs or additional pipeline stages, we expect the verification time to increase linearly. Compared to Rebecca, which is limited to the verification of pure hardware implementations, the hardware/software co-verification approach of Coco employs more aggressive optimization measures by simplifying correlation sets through concrete values from the execution trace, and can therefore more easily deal with scalability issues.

5.4. Practical Evaluation

The purpose of Coco is to verify the security of masked software implementations at the level of gate-level netlists of the underlying hardware. The main application for the tool are ASIC designs of processors, where Coco allows to perform a verification of the final netlist of a design before tape-out. The fabrication of an ASIC is clearly beyond the scope of this paper. However, in order to show that our approach indeed leads to secure implementations in practice, in this section we map a sample of a verified netlist to an FPGA and perform an empirical analysis.

Several things need to be considered when doing this mapping. When syn-

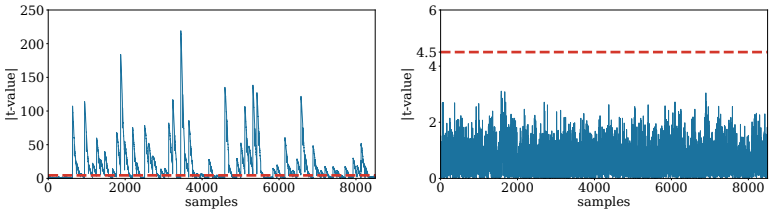


Figure 5.: T-test scores of the original (left) and the secured (right) register file during the execution of a first-order DOM Keccak S-box using 100 000 power traces.

thesizing hardware designs for FPGAs, the resulting netlist does not contain typical CMOS building blocks but rather, among others, lookup tables (LUTs) that are configured to match the original hardware design on a logical level but not on netlist-level. This is especially problematic since FPGA synthesis tools tend to merge multiple logic gates into single, typically 3 to 6-bit LUTs. The resulting hardware will still be equivalent from a pure logic perspective, however, certain characteristics such as the strict separation of registers in our secured register file can get lost in the translation process. Therefore, we manually map the ASIC netlist of the original and the secured Ibex core to FPGA netlists that match the ASIC netlists as closely as possible. This step involves, amongst others, ensuring that every logic gate is represented by a single dedicated LUT. Since this process is mostly manual, and thus very time consuming, we decided to focus our leakage assessment only on the most important parts of the secured Ibex which are needed to execute cryptographic implementation: the register file and a simple ALU.

In our experiments, we compare the execution of a masked Keccak S-box computation using (1) the basic register file as it can be found in the original Ibex core, (2) the secured register file including (one-hot encoded) gated reads and writes (cf. Section 3.2). Following the guidelines of Goodwill et al. [Goo+11], we use Welch's t-test to show practical first-order protection of first-order masked software implementations. The basic idea is to measure the significance of the difference of means of two distributions by constructing two trace sets, one with random inputs and one with constant inputs. In the case of a masked implementation it means that the secret, native inputs are fixed, while the masks and shares are generated randomly. The null-hypothesis is that both trace sets have equal means, i.e., they cannot be distinguished from each other. The null-hypothesis is rejected with a confidence greater than 99.999% if the absolute t-score t stays below 4.5.

For our experiment, we execute the register-only (*reg.*) variant of the DOM first-order masked Keccak S-box, as introduced in Table 3. In order to measure

the power consumption, we use the SAKURA-G board [GIS14] equipped with a Xilinx Spartan-6 FPGA. We connect the board to a PicoScope 6404C at 312.5 Ms/s sampling rate, the Ibex components operate at a clock frequency of 8 MHz.

Figure 5 shows the results of our leakage assessment using 100 000 traces. The left presents the t-test results for the original, unprotected register file during the execution of the first-order DOM Keccak S-box. As expected, the t-test shows significant peaks over the 4.5 border which indicates first-order side-channel leakage. The right presents the t-test results for the same code when running on our secured version of the register file. Here, the leakage assessment reveals no significant peaks, which indicates that our secured design works as expected.

6. Conclusion

In this paper, we presented Coco, the first tool for co-design and co-verification of masked software implementations on CPUs. Coco takes a CPU netlist, together with a masked assembly implementation, and then formally verifies its leakage-free execution down to the gate-level. While previously presented software verification approaches mainly work on algorithmic level and model only a few select CPU side-effects, Coco can detect any CPU design aspect that could reduce the protection order of masked software implementations.

We show the practicality of our work, by analyzing the popular 32-bit RISC-V Ibex core with Coco. We detect various design aspects that reduce the protection order of our tested software implementations and propose respective fixes, mostly in hardware. Our resulting secured Ibex core has an area overhead of about 10%, the runtime of software on this processor is largely unaffected, and the formal verification with Coco of an, e.g., first-order masked Keccak S-box running on this core takes around 156 seconds. We demonstrate the effectiveness of the proposed design modifications in a practical evaluation on an FPGA.

Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and comments, which helped in improving the paper. This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", and the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, via the Fermion project (grant nr 867542), and via the project IoT4CPS. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Appendix A. Fourier Expansions of Boolean Functions

$$
\begin{aligned}
\textsc{And} \quad & W(a \wedge b) = \frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ab \\
\textsc{Or} \quad & W(a \vee b) = -\frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b + \frac{1}{2}ab \\
\textsc{Xor} \quad & W(a \oplus b) = ab \\
\textsc{Xnor} \quad & W(\overline{a \oplus b}) = -ab \\
\textsc{Not} \quad & W(\neg a) = -a \\
\textsc{Mux} \quad & W(c \ ? \ a : b) = \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ac + \frac{1}{2}bc
\end{aligned}
$$

Appendix A1. AND Gate

$$W(c?a : b) = p_0 + p_1 \cdot a + p_2 \cdot b + p_3 \cdot ab$$

$$
\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}
$$

$$
\Rightarrow \quad \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{pmatrix}
$$

Appendix A2. Multiplexers

$$
\begin{aligned}
W(c?a : b) = & p_0 + p_1 \cdot a + p_2 \cdot b \\
& + p_3 \cdot c + p_4 \cdot ab \\
& + p_5 \cdot ac + p_6 \cdot bc \\
& + p_7 \cdot abc
\end{aligned}
$$

We can build a an equation system using all possible input combinations for the variables a, b, and c and then solve for the unknown coefficients p_0 to p_7 as shown below.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \\ 0 \\ 0 \\ -0.5 \\ 0.5 \\ 0 \end{pmatrix}$$

Appendix B. Verification Flow of Coco

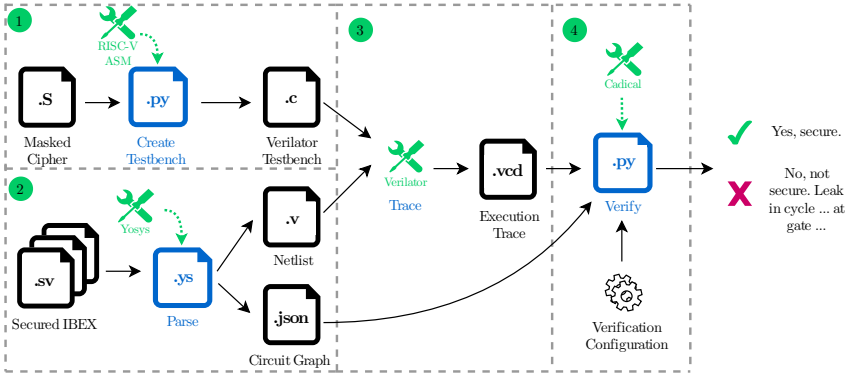


Figure 6.: Verification flow of Coco. The workflow consists of four steps, the creation of the testbench (1), parsing (2), trace (3) and verification(4). In the end, Coco either confirms that the execution is secure or points out the flaw(s) in a specific gate, in a specific cycle.

References

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and*

Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. Vol. 8968. Lecture Notes in Computer Science. Springer, 2014, pp. 64–81.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 457–485.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[Bar+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 189–228.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 397–426.

[Bel+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In:

EUROCRYPT (3). Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–341.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[BP12] Joan Boyar and René Peralta. "A Small Depth-16 Circuit for the AES S-Box". In: *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 287–298.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 69–81.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[ETH] ETH Zurich. *PULP Platform*. https://pulp-platform.org/. Retrieved on September 15th, 2020. URL: https://pulp-platform.org/.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[Gao+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Hung Pham, and Francesco Regazzoni. "An Instruction Set Extension to Support Software-Based Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 283–325.

[GIS14] Hendra Guntur, Jun Ishii, and Akashi Satoh. "Side-channel AttacK User Reference Architecture board SAKURA-G". In: *IEEE 3rd Global Conference on Consumer Electronics, GCCE 2014, Tokyo, Japan, 7-10 October 2014*. IEEE, 2014, pp. 271–274.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A testing methodology for side-channel resistance validation". In: *NIST Non-Invasive Attack Testing Workshop*. 2011.

[Gro+16a] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. "Bitsliced Masking and ARM: Friends or Foes?" In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*. Vol. 10098. Lecture Notes in Computer Science. Springer, 2016, pp. 91–109.

[Gro+16b] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs". In: *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*. Vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. “Higher-Order Side-Channel Protected Implementations of KECCAK”. In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 2017, pp. 205–212.

[Har+03] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. “PINPAS: A tool for power analysis of smartcards”. In: *International Conference on Information Security (SEC2003)*. 2003, pp. 453–457.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. “Private Circuits: Securing Hardware against Probing Attacks”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KS20] Pantea Kiaei and Patrick Schaumont. “Domain-Oriented Masked Instruction Set Architecture for RISC-V”. In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 465.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. “SILVER - Statistical Independence and Leakage Verification”. In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[low19] lowRISC contributors. *Open Titan*. https://opentitan.org/. Retrieved on March 23th, 2023. 2019. URL: https://opentitan.org/.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[MGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. “Protecting RISC-V against Side-Channel Attacks”. In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 45. DOI:

10.1145/3316781.3323485. URL: https://doi.org/10.1145/3316781.3323485.

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1297.

[Moo+19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. "Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 256–292.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 199–216.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches". In: *J. Cryptol.* 24.2 (2011), pp. 292–321.

[ODo14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5.

[Opea] OpenHW Group. *CV32E40P*. https://github.com/openhwgroup/cv32e40p, Retrieved on September 15th, 2020. URL: https://github.com/openhwgroup/cv32e40p.

[Opeb] OpenHW Group. *CVA6*. https://github.com/openhwgroup/cva6. Retrieved on September 15th, 2020. URL: https://github.com/openhwgroup/cva6.

[PS08] Andrei Pavlov and Manoj Sachdev. *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies: Process-Aware SRAM Design and Test*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 1402083629.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. “Mind the Gap: Towards Secure 1st-Order Masking in Software”. In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 282–297.

[QS01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards”. In: *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. Ed. by Isabelle Attali and Thomas P. Jensen. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001, pp. 200–210.

[Reg+09] Francesco Regazzoni, Alessandro Cevrero, François-Xavier Standaert, Stéphane Badel, Theo Kluter, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. “A Design Flow and Evaluation Framework for DPA-Resistant Instruction Set Extensions”. In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 205–219.

[Ren+11] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. “A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices”. In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 109–128.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. “Consolidating Masking Schemes”. In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[She+21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. “Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers”. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[Sny22] Wilson Snyder. *Verilator*. https://www.veripool.org/wiki/verilator. Retrieved on February 2nd, 2021. 2022.

[TM04] Sarah Thompson and Alan Mycroft. “Abstract Interpretation of Combinational Asynchronous Circuits”. In: *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. Vol. 3148. Lecture Notes in Computer Science. Springer, 2004, pp. 181–196.

[Tri03] Elena Trichina. “Combinational Logic Design for AES SubByte Transformation on Masked Data”. In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 236.

[Wol16] Claire Wolf. *Yosys Open Synthesis Suite*. http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021. 2016.

4

Secure and Efficient Software Masking on Superscalar Pipelined Processors

Publication Data. Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Masking on Superscalar Pipelined Processors".
In: *ASIACRYPT*. 2021.

Contribution. The author of this thesis contributed to the concepts, performed all the experiments, made the implementations and wrote large parts of the text.

Secure and Efficient Software Masking on Superscalar Pipelined Processors

Barbara Gigerl[1], Robert Primas[1], Stefan Mangard[1,2]

[1] Graz University of Technology [2] Lamarr Security Research

Abstract Physical side-channel attacks like power analysis pose a serious threat to cryptographic devices in real-world applications. Consequently, devices implement algorithmic countermeasures like masking. In the past, works on the design and verification of masked software implementations have mostly focused on simple microprocessors that find usage on smart cards. However, many other applications such as in the automotive industry require side-channel protected cryptographic computations on much more powerful CPUs. In such situations, the security loss due to complex architectural side-effects, the corresponding performance degradation, as well as discussions of suitable probing models and verification techniques are still vastly unexplored research questions.

We answer these questions and perform a comprehensive analysis of more complex processor architectures in the context of masking-related side effects. First, we analyze the RISC-V SweRV core — featuring a 9-stage pipeline, two execution units, and load/store buffers — and point out a significant gap between security in a simple software probing model and practical security on such CPUs. More concretely, we show that architectural side effects of complex CPU architectures can significantly reduce the protection order of masked software, both via formal analysis in the hardware probing model, as well as empirically via gate-level timing simulations. We then discuss the options of fixing these problems in hardware or leaving them as constraints to software. Based on these software constraints, we formulate general rules for the design of masked software on more complex CPUs. Finally, we compare several implementation strategies for masking schemes and present in a case study that designing secure masked software for complex CPUs is still possible with overhead as low as 13%.

1. Introduction

Cryptographic primitives are primarily designed to withstand mathematical attacks in a black-box setting. However, as soon as these primitives are deployed in the real world, they find themselves in a grey-box setting in which an attacker may observe additional physical side-channel information, such as instantaneous power consumption that can be used to extract secrets like cryptographic keys. One particularly powerful example of such a side-channel attack, differential power analysis (DPA), was introduced in 1999 by Kocher et al. [KJJ99]. In

this type of attack, the adversary observes a device's power consumption while encrypting several known plaintexts, and can then extract sensitive information using statistical analysis.

The typical approach of protecting against these attacks is to implement algorithmic countermeasures, like masking [Bar+17; Bel+17; Cnu+16; GM17; ISW03; Rep+15]. The main idea of masking is to make computations independent from the actually processed data. For this purpose, masking schemes split input and intermediate variables of cryptographic computations into $d + 1$ random shares such that observations of up to d shares do not reveal any information about the native (unmasked) value. The security of such dth-order protected computations relies, amongst others, on the assumption of independent leakage, i.e., independent computations result in independent leakage [Ren+11]. However, many academic works in the past have shown that such assumptions are typically not satisfied on ordinary CPUs, for example, memory transitions in the register file or RAM can leak the Hamming distance between two shares [Bal+14; Cor+12; Gro+16; MMT20; PV17]. In general, one can work around these problems using two different strategies. Works like [Bar+21; Gig+21; Gro+16; PV17] show that one can design dedicated masked software implementations that take specific characteristics of the microprocessor into account, e.g., by never processing shares of the same variable in immediate succession. Alternatively, one can follow the *lazy engineering* approach, accept a certain loss of masking protection order due to architecture side-effects and compensate for that by using a protection order that is higher than theoretically required. This strategy was more formally analyzed by Balasch et al. [Bal+14] who also formulated the so-called order reduction theorem. This theorem states that, when considering simple register-based CPU architectures, the security of a dth-order masked software implementation reduces to $\lfloor \frac{d}{2} \rfloor$-th order if transition-based leakage is taken into account.

Building efficient and correct masked software implementations is generally difficult since one either needs to (1) carefully patch implementations for specific microprocessors [Bar+21; Gro+16; PV17], or (2) invest in masking orders that are a lot higher than required [Bal+14]. In both cases, the runtime of the resulting masked software implementations is significantly increased and subsequent manual leakage assessments are needed to confirm that the performed modifications have the desired effect, which is a quite labor-intensive and error-prone task. This situation becomes only ever more difficult with increasing processor complexity. For example, the effects of multiple ALU pipeline stages, forwarding logic, superscalar building blocks, caches, and complex logic for handling loads/stores on masked software implementations have not been analyzed in this detail before. One reason for that might be the sheer complexity of application-level processors that usually consist of superscalar building blocks and multi-stage pipelines. On such processors, identifying and understanding masking related side-effects can barely be done manually anymore. Here, automated analysis methods that can give concrete conditions under which masked software implementations can

guarantee a certain protection order on such CPUs are more relevant then ever.

In this context, a recent work by Gigerl et al. [Gig+21] studies the simple Ibex core with Coco, a tool that can verify the correct execution of masked software implementations on given CPU netlists, while considering all possible architectural side effects. Simply speaking, Coco treats an entire CPU design as a hardware circuit and then tracks all the shares of executed masked software implementations over several cycles using methods that are inspired by Rebecca [Blo+18]. One result of their analysis is a slightly modified *secured* Ibex core on which masked software implementations can preserve their theoretic protection order in practice if a few simple software constraints are followed. While this result is certainly interesting for applications like smart cards where low computing power is sufficient, many other IoT or automotive use cases require the usage of significantly more powerful processors. This raises a number of questions about the performance, as well as the theoretic and practical security of masked software on more complex CPUs.

Our contribution We answer these questions by providing the following contributions:

- We generate several generic higher-order masked cryptographic software implementations using *Tornado* and show with Coco that there is little hope that such implementations can even provide 1st-order protection on more complex CPU cores. We demonstrate this based on the dual-issue 9-stage RISC-V SweRV core.
- In addition to the formal analysis of Coco, we perform gate-level simulations to demonstrate that architecture-based glitch effects are visible in practice and reduce the security of masked software by multiple orders. This points out a significant gap between security in the simple software probing model and practical security, and further motivates the verification of masked software on concrete CPU netlists in a more hardware focused probing model.
- We then further analyze the components of SweRV that do not exist in simpler cores, identify new problems, and discussed possible solutions in software or hardware.
- Based on this analysis, we formulate more general rules for designing masked software that takes into consideration properties such as the pipeline length, the amount of execution units, or architectural buffers. We also present arguments why relying on the *lazy engineering* approach alone, as proposed by [Bal+14], does not seem viable anymore in case of more complex CPUs.
- Finally, we present a case study that compares how efficiently our derived software constraints can be met with different implementation strategies.

Maybe somewhat surprisingly we show that, with knowledge about a processors netlist, one can build secure and efficient masked software for SweRV-like cores with overhead as low as 13%.

Outline In Section 2 we cover relevant background on masking and the verification of masking, including the basic working principles of Coco and *Tornado*. In Section 3, we describe the evaluation setup for the analysis of more complex CPUs with Coco, present some initial verification results and describe the significance of these in a practical evaluation. In Section 4, we present a detailed analysis of SweRV architecture, describe all hardware components that can pose problems to masked software implementations and propose viable solutions. In Section 5, we list the generic software constraints and evaluate their overhead in Section 6. We conclude our work in Section 7.

Open Source We plan to publish both, our modified SweRV core, as well as the corresponding software implementations that are used in this paper on github [1].

2. Background

2.1. Masking

Masking has become one of the first-choice measures to defeat power-analysis attacks on algorithmic level. In general, masking is a secret-sharing technique which splits intermediate values of a computation into $d+1$ shares. The shares are uniformly random, such that an attacker who observes up to d shares cannot infer any information about the underlying native value. A dth-order Boolean masking scheme splits a native variable s into $d+1$ random shares $s_0 \dots s_d$, such that $s = s_0 \oplus \dots \oplus s_d$. The values $s_0 \dots s_{d-1}$ are chosen uniformly at random while $s_d = s_0 \oplus \dots \oplus s_{d-1} \oplus s$. Consequently, each share s_i is uniformly distributed and statistically independent of the native value s.

Implementing linear functions when designing masked cryptographic implementations is trivial, as they can simply be computed on each share individually. However, non-linear functions (S-boxes) are not as simple, since computations involve multiple shares of a native value at the same time, which is more difficult to implement in a secure and correct manner. Therefore, the main interest in literature lays on masked implementations of non-linear functions [Bar+17; Bel+17; Cnu+16; GM17; GMK16; ISW03; Rep+15]

2.2. Formal Verification of Masking

In general, masked implementations must ensure that each intermediate value of a computation is statistically independent of any native values. The verification

[1] https://github.com/barbara-gigerl/sw-masking-swerv

of this property is usually done with the help of a security model that specifies the abilities of an attacker. Typically, it is assumed that the ability of the attacker is to place a certain amount of probes in a computation, that allow monitoring concrete values at those locations.

Formal Verification of Hardware Implementations. The *classical probing model* by Ishai et al. [ISW03] is the most commonly used security model for masked hardware circuits and it's accuracy in modeling real world attacks has been confirmed by many works [Fau+10; RP10]. Here, an attacker is allowed to place up to d probes at any location in a circuit, which can be used to observe the corresponding gates/wires permanently. A masked hardware implementation is considered dth-order secure if an attacker cannot learn any information about the native values by combining all d observations. Examples of tools that can verify classical probing security for cryptographic hardware implementations are are Rebecca [Blo+18], Silver [KSM20], and maskVerif [Bar+19]. These tools are mainly tailored to the verification of masked hardware (ASIC/FPGA) implementations. maskVerif does offer some support for software implementations but (1) can only deal with code that is written in a special intermediate language, and (2) only considers simple CPU side-effects such as register overwrites.

Formal Verification of Software Implementations. On software side, the research community has also published many methods and tools to automatically generate or verify masked software implementations [Bar+15; Bar+16; Bay+13; EWS14; Mos+12; Zha+18]. More recently, Belaïd et al. proposed *Tornado* [Bel+20], a tool that takes a high-level description of an unmasked cryptographic function, generates a corresponding (any-order) masked C implementation, and verifies its probing security. *Tornado*'s verification itself is based on tightPROVE+, an extension of tightPROVE [Bel+17]. tightPROVE+ performs the verification of masked software in the *register probing model*. This model allows an attacker to place probes on individual words of a processor's register file, and to use them for one cycle each during the execution of a masked software implementation. Hereby, it is assumed that the probed registers cause independent leakage, in other words, no additional potential side effects of a processors architecture, such as glitches or register overwrites, are considered [Ren+11].

More precise verification tools, that e.g. also cover transition leakage have been presented in [Ath+20; Bar+21; WSW19], while with Coco, Gigerl et al. have recently presented a tool that can verify the correctness of masked software implementations while considering possible architectural side effects of a given CPU netlist [Gig+21].

2.3. Coco

Coco is a tool for the co-design and co-verification of masked software implementations on CPU netlists [Gig+21]. It formally verifies the security of (any-order)

masked assembly implementations that are executed on concrete CPUs, defined by gate-level netlists. Coco's verification strategy is inspired by Rebecca but extended in a way such that the verification of masked software, when running on hardware, is converted into a pure hardware verification problem. This involves not only the addition of control-flow awareness but also several performance improvements since entire CPUs are usually significantly larger designs, when compared to pure hardware implementations of cryptographic functions. Coco does not only capture transition-based effects, but in principle any glitch-related hardware side-effects that can be derived from a CPU netlist. This is also formalized in the so-called *time-constrained probing model*, in which an attacker can use each probe to measure any specific gate/wire for the duration of one clock cycle that can be chosen independently for each probe.

Verification Flow In the following, we briefly outline the workflow of Coco, broken into multiple steps. Steps **A** and **B** explain how the execution of software can be combined with an otherwise purely hardware-focused verification method. Step **C** then describes the application of Coco in a bit more detail.

A. Yosys [Wol16] is used to parse the given CPU design into a gate-level netlist. The masked assembly implementation together with the netlist is then given to Verilator [Sny22], which produces a cycle-accurate simulation of the execution in form of an *execution trace*. The execution trace contains concrete values of all CPU control signals during the software execution.

B. Registers or memory locations in the CPU netlist receive annotations (labels) that indicate the location of shares and randomness at the start of the software execution.

C. The CPU netlist, execution trace, initial labeling and desired verification order is given to the verifier, which propagates the labels through the CPU netlist, for as many cycles as the software execution takes. In case Coco detects that a specific gate in the netlist leaks information about a native value (by observing a combination of shares of the same native value), e.g. due to implementation mistakes or architectural side-effects, the exact gate in the netlist and the execution cycle is reported as a leak. For a more detailed description of this verification method we refer to the original publication [Gig+21].

2.4. RISC-V SweRV Core

The SweRV processor family [Ted19] was first introduced by Western Digital in 2019 and designed for data-intensive applications like storage controllers and industrial IoT. As of today, there are three different variants of the processor: the EH1, the EH2 and the EL2 [Wes19]. The EH1 features a 32-bit superscalar 9-stage pipeline, while the EH2 basically adds a second thread with a dedicated

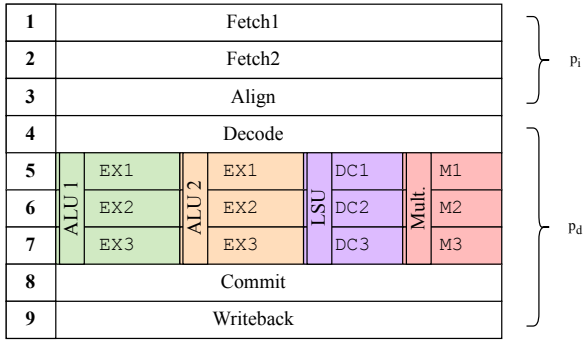


Figure 1.: Pipeline stages of SweRV [Wesa]

register file and instruction fetch buffer. The EL2 is a smaller version of the EH1 with only 4 pipeline stages and one execution unit.

In our experiments, we use the SweRV EH1 core[2], which implements the RISC-V RV32IMC instruction set and has nine pipeline stages [Wesa], as sketched in Figure 1. The first three pipeline stages (Fetch1, Fetch2, Align) are responsible for loading instructions from the instruction memory and storing them into the fetch buffer. In the Decode stage, the instructions are decoded and prepared for execution. The execution happens in pipeline stages 5-7, either in the Load-Store Unit (DC1,DC2,DC3), the multiplication unit (M1, M2, M3) or the ALUs (EX1, EX2, EX3). The EH1 core has a dual-issue pipeline, which means that in each clock cycle, the processor can decode two instructions and send them to two different ALUs. In the last two pipeline stages, Commit (EX4) and Writeback (EX5), the final result is stored in the register file. There are several peripherals attached to the core via an AXI4 bus, including the SRAM and instruction and data closely-coupled memories. The core operates in-order, except for loads which might get executed earlier when the value is needed in the pipeline.

According to Western Digital, the SweRV EH1 core can be operated at frequencies of up to 1 GHz [Wesb] and its performance can be compared to an ARM Cortex A15, making it outperform other RISC-V processors like the Berkely BOOM core [The]. This makes EH1 an interesting target to analyze the effects of more complex CPU architectures on masked software implementations. Another reason why we chose SweRV EH1 is Coco's current requirement of CPU designs to be written in Verilog or SystemVerilog.

[2] https://github.com/chipsalliance/Cores-SweRV

3. Generic Masked Software on SweRV

In this section we perform an initial analysis of generic (higher-order) masked software implementations on the SweRV EH1 core and show that, even after applying the same hardware modifications as proposed for Ibex in [Gig+21], a more complex CPU architecture introduces additional problems that can reduce the protection of masked software by serveral orders. In Section 3.1, we describe a few small hardware modifications that we carry over from Gigerl et al.'s *secured* Ibex to SweRV, that would otherwise lead to identical problems on SweRV. In Section 3.2 we describe modifications we made to Coco's verification flow itself so that it can better handle CPU designs that are significantly larger than Ibex. In Section 3.3, we generate generic, up to 4th-order masked software implementations of the Keccak S-box using *Tornado*, verify their execution on the *secured* SweRV using Coco, and conclude that there is little hope that such implementations can achieve even just 1st-order protection. Finally, we present additional empirical evidence of the impact of architectural glitches on masked software via several gate-level timing simulations in Section 3.4.

3.1. Modifications of SweRV

Gigerl et al. have analyzed the simple 32-bit RISC-V Ibex core in terms of software masking-related side effects. As a result of their analysis, they pointed out three hardware components that can cause unintended combinations of shares during the execution of masked software implementations that are completely invisible from software perspective: the register file, the Arithmetic Logic Unit (ALU), and the Load-Store Unit (LSU). Not surprisingly, the SweRV core has similar problems, which is why we briefly discuss how we map these proposed hardware fixes from Ibex to the SweRV core in the following. The resulting *secured* SweRV core will then serve as the base of our further analysis. We expect that the total area overhead of the hardware modifications for the SweRV core is very similar to the Ibex core as analyzed in [Gig+21], which was about 2 kGe. Since the SweRV core is much larger, this overhead is insignificant.

We use SweRV core commit `499378d0c67ab11965` as the baseline for our modifications. For our analysis, we disable closely-coupled memories for instructions and data, but enable the instruction cache. We do this since (1) the instruction cache is large enough to hold all implementations that we intend to test, (2) we want to analyze the "worst-case" in which the CPU can fetch instructions without delay, thereby achieving the maximal possible amount of instructions (and side-effects) in the pipeline stages. Hence, when running a verification with Coco, we execute each software implementation twice, once to load it into the instruction cache from instruction memory, and once to perform the actual verification.

Register File Ordinary register file implementations consist of a group of register words (32×32bit for RV32IMC) plus addressing logic for reading two words and writing one word within one clock cycle. This addressing logic is usually implemented via multiplexer trees that select source and destination registers depending on the currently decoded instruction. As previously shown for Ibex, these selector signals are usually calculated by combinatorial logic within the same cycle as the actual read/write event. Consequently, within a single clock cycle, differences in signal propagation delays can cause glitches on these selector signals, which in return can cause a read/write port to unreliably switch between multiple register words until the selector signals at all multiplexers are stable[3]. This is problematic for masked software implementations as they hold many shares in the register file that must be kept strictly separated from each other.

The proposed solution for this problem is to replace multiplexer trees with OR trees while introducing a one-hot encoded gating mechanism for each value that is calculated in the previous clock cycle and buffered in a additional register [Gig+21]. This mechanism ensures that glitches on a read/write port can only ever happen between the operand of two consecutive instructions. In the SweRV core, we face the same problems and fix these by applying the same register gating concept. The main difference here is the fact that SweRV features four read and three write ports, compared to Ibex's two read and one write port. Gating the read and write ports for SweRV works almost straightforward, except for the third write port, which is used for data from the memory, and requires a dedicated solution (cf. subsubsection 4.2).

Concurrent ALU Computations Cores like Ibex and SweRV always concurrently calculate simple operations like AND, XOR, ADD, SHIFT in the execution stage and later only forward the result that is actually needed by the currently executed instruction. This is not a problem for most masking techniques, however there do exist some masking techniques that store individual shares of the a native value in the same register word[Bar+17]. This is okay as long as all computations keep the individual bits of a register word separate from another, e.g., by performing only bit-wise operations such as AND and XOR. Operations such as ADD or SHIFT on the other side do combine bits of individual operands and can thus create side-channel leakage, even if the results of these computations are ultimately discarded.

The suggested solution for the Ibex core is to implement a gating mechanism that ensures that only the intended computation is performed. This mechanism can also be easily carried over from Ibex to SweRV.

[3]Even if the selector signals were stable, e.g. by calculating and buffering them in the previous clock cycle, there is still no guarantee that this signal arrives at all multiplexers in stable condition in the next clock cycle due to different wire lengths.

Data Memory Storing shares in the data memory leads to similar problems with glitches in the addressing logic as for the register file. In theory, one could again use the same one-hot encoded gating mechanism as discussed before, however, this approach does not scale well for the large address ranges that are required for data memory. Consequently, Gigerl et al. propose a trade-off that consists of using only partially one-hot encoded addresses for data memory which can be implemented with an area overhead that is indeed negligible when compared to the area of SRAM blocks themselves. The downside of this trade-off is that only memory words within certain address ranges (blocks) are properly separated from each other. This is sufficient as long as a block is large enough to hold all the shares that need to be kept isolated from each other during the execution of masked software implementations.

We apply the same LSB one-hot address encoding to SweRV's data memory. Since the SweRV core reads 64 bit from the memory in one cycle instead of 32-bit, we gate memory words on 64-bit granularity.

3.2. Modifications of Coco

In this section we briefly outline modifications that we have made to Coco's verification workflow so that it can better handle large CPU designs. These modifications first and foremost reduce SweRV's circuit size which in return also significantly reduces Coco's verification runtime.

Removal of Unused Logic As mentioned before, we ensure that instructions can be directly loaded from the instruction cache during Coco's verification. We only ever use the slower instruction memory in a read-only fashion to fill the instruction cache and can, for the pure purpose of Coco's verification, remove any unused logic that would allow writes to data memory, which reduces the circuit size by about 29%.

Control Wire Tagging The initial version of Coco effectively treats each wire of a CPU netlist equally and does not distinguish control from data wires. In reality, only a small fraction of wires can actually affect the data that is processed by a masked software implementation in such a way that side-channel related problems could occur. Therefore, we adapted Coco such that it is possible to tag wires as explicit *control* signals. During the verification, Coco will then simply ignore these wires instead of applying the laborious process of constructing empty SAT equations for them. Clearly, this tagging needs to be done carefully such that we do not later overlook any architecture side-effects during Coco's verification. Since manual tagging of individual wires is infeasible for entire CPU designs, we instead only do this in a course-grained manner and only in cases where we can easily deduce that there will be no consequences for the processed data of software with constant (data-independent) control flow. More precisely,

we tag the instruction memory, instruction cache and signals depending solely on those as control signals automatically.

3.3. Initial Analysis of the SweRV Core

In this section we present our initial analysis of several higher-order masked software implementations on the *secured* SweRV core that already includes all hardware modifications that were proposed in the previous analysis of Ibex [Gig+21]. First, we use *Tornado* to generate generic, up to 4th-order masked C implementations of the Keccak S-box that are formally verified in *Tornado*'s register probing model, meaning that an attacker observing up to d intermediate values (of the algorithm) is not allowed to learn information about native values. We then analyze the execution of these implementations on SweRV using Coco to get an impression of how many more issues can be detected in Coco's time-constrained probing model, in which an attacker, able to observe up to d wires/gates in the CPU netlist throughout one clock cycle each, is not allowed to learn information about native values.

Since Coco can only deal with assembly implementations by default, we create an assembly wrapper around the *Tornado*-generated C functions and adapt the work flow accordingly. We then analyze these implementations using Coco, while targeting the verification of 1st-order protection. Unfortunately, the verification results show that none of the tested implementations can even reach just 1st-order protection. Upon first inspection of the reported problems, we can see that multiple additional issues still exist within SweRV that can significantly reduce the protection order of our tested software implementations. For example, the forwarding logic in SweRV's 9-stage pipline is reported by Coco as one of the main culprits for the loss of multiple protection orders in the time-constrained probing model.

3.4. Empirical Evaluation

In order to empirically confirm the problems identified by Coco in the SweRV core, we perform and analyze gate-level simulations in this section. More concretely, we perform gate-level timing simulations of the forwarding logic within SweRV's pipeline (see Figure 3) using multiple cell libraries to better illustrate how problems in the time-constrained probing model can relate to practical problems. Our evaluation reveals that glitches in the forwarding logic can lead to independent occurrences of up to five shares on one wire within one clock cycle, and combined occurrences of up to three shares at the same time. We note that while the exact behaviour of glitches strongly depends on the used standard cell library, all of our tested standard cell libraries report leaks leading to a reduction in the masking order between three and five.

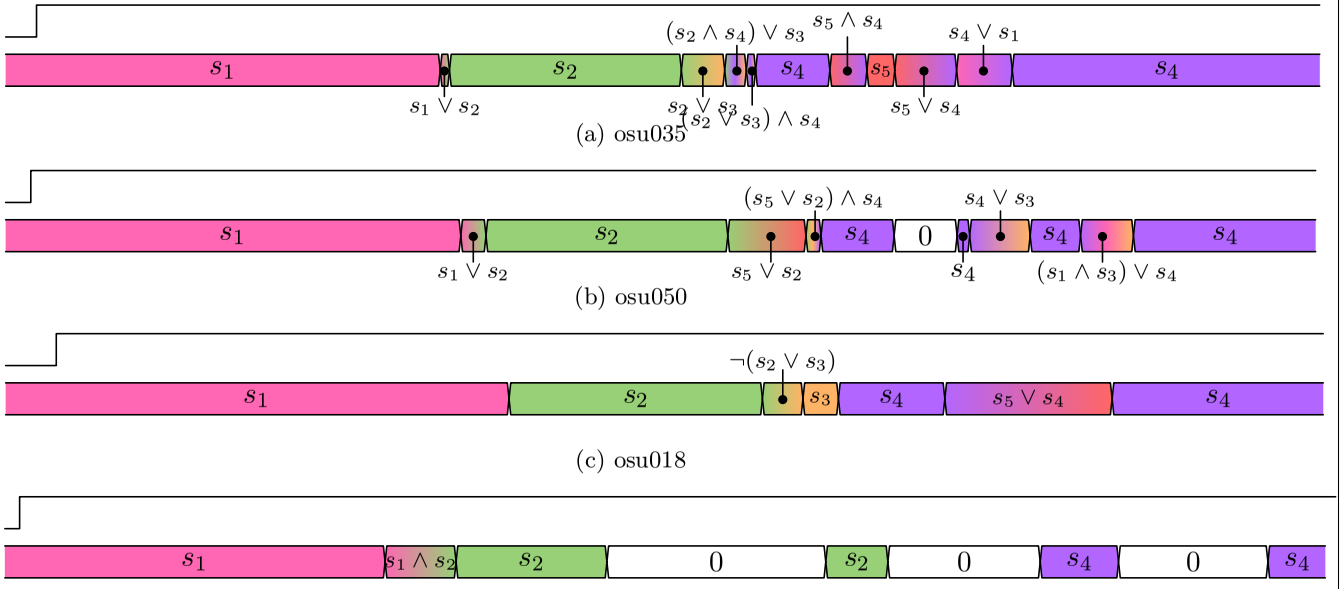


Figure 2.: Value of wire `fwd_data` at the beginning of a clock cycle for four different standard cell libraries. $s_1 \ldots s_5$ denote up to five of the shares that are visible in this experiment due to architectural glitch effects. Since the analyzed time window in each plot is different (due to different propagation delays) we have applied suitable horizontal scalings to improve readability. The time before the rising clock edge is always 10ps.

Setup We use signal traces from the post-synthesis simulations of the SweRV core netlist. The synthesis process maps logic gates in the netlist to suitable cells in the standard cell library, which defines the exact behavior and delay of each cell. We investigate and compare four different open-source cell libraries[4], osu035, osu018, osu050, and gscl45nm. The mapping process is performed by Yosys[Wol16], before running the simulation with Modelsim to obtain an execution trace of our test program.

The same test program is used in all four evaluation scenarios. The test program works with a native value split into 10 shares, which corresponds to a 9th-order masked implementation. First, the test program executes 10 instructions, each operating on exactly one share. This effectively stores each share to its own register in a specific pipeline stage. Second, the test program executes an instruction referring to a previously computed result, which sends the shares in the pipeline registers to the bypass logic, which finally forwards the correct share to the ALU. It should be noted that the program is correctly masked on algorithmic level because exactly one share is processed per instruction.

Results Figure 2 shows what information an attacker can observe by probing the wire `fwd_data` in SweRV's forwarding logic for the duration of one clock cycle using different cell libraries. Each plot additionally shows the corresponding clock signal and contains marks that indicate at which point in time a specific share (or combination of shares) is visible until the value of the wire has stabilized. Since the analyzed time window in each plot is different (due to different propagation delays) we have applied suitable horizontal scalings to improve readability.

From these plots we can see that an attacker can always observe at least three shares (Figure 2d), and at most five shares (Figure 2a-c) within one clock cycle when probing the `fwd_data` wire. Sometimes, shares do not appear independently, but also in combination with other shares. For example, in Figure 2a, the attacker first observes s_1, and then s_1 in combination with s_2. Note that both, the occurrence of multiple shares independently within one cycle, or the occurrence of combinations of shares at any point in time breaks the assumption of independent leakage.

Clearly, this evaluation is not exhaustive. Every technology, every cell library, and every different placement of a design, leads to different timing properties and differences in the exact leakage. Also concrete ASIC or FGPA prototypes are just instances of particular configurations. The exact quantification of the leakage, i.e. determining the number of traces that are needed for exploitation in a particular configuration, is not in the scope of this paper. In fact, it is also not clear if it would be possible to find a representative configuration and setup that would allow more than making a statement on leakage for one particular realization in one particular setup. A worst case setup would be a library with

[4] https://github.com/RTimothyEdwards/qflow/tree/master/tech

delay settings that lead to the observation even all 10 shares in a single clock cycle.

Instead of focusing on more specific instances, the focus of our analysis in this section was on showing that problems identified using Coco actually lead to critical signal transitions in the design. Given the empirical confirmation of critical signal transitions, we therefore use Coco as a reference for the identification of critical design elements in the design of SweRV. With Coco we are able to formulate a generalized statement about the security of a masked software implementation in the time-constrained probing model, which is independent of a specific technology or platform.

4. Analysis of Problems on SweRV

As shown in our previous analysis of generic (higher-order) masked software implementations, the hardware components of more complex CPUs can cause a significant reduction in the protection order. In this section, we discuss these problematic components in the *secured* SweRV core that already has the Ibex-patches applied (cf. Section 3.1). We divide these problems into ***big*** and ***small*** problems, based on how many shares may be combined, since, as we show later in Section 5, one can follow different strategies to deal with them. A component causes a ***big*** problem when more than two shares can be potentially combined. A ***small*** problem indicates that a component can combine at most two shares. For each potential leakage source, we discuss the options of making further modifications in hardware or shifting this problem as a constraint to masked software implementations.

4.1. Pipelines and Execution Units

The dual-issue SweRV EH1 core features nine pipeline stages and can process two instructions per clock cycle. Accordingly, the fetch/decode stages (1-4) can handle multiple instructions at the same time, the execution/writeback stages (5-9) exist twice, while the lesser used multiply (5-7) and load/store stages (5-7) exist only once (c.f. Figure 1). The dual issue design also requires a register file with four read ports and three write ports. Since symmetric cryptographic software implementations are usually implemented with constant (data independent) control flow, which is also the case for all our tested software implementations, only the later execution/writeback stages (5-9) get in touch with actual data and can thus cause potential side-channel related problems.

A typical optimization in pipelined CPU designs is the usage of forwarding logic, also known as bypass-logic, that can redirect the result of an instruction from a later pipeline stage to a previous stage without needing to wait for the result to be written into the register file. Forwarding significantly reduces the occurrence of pipeline stalls in cases where one instruction uses an operand that

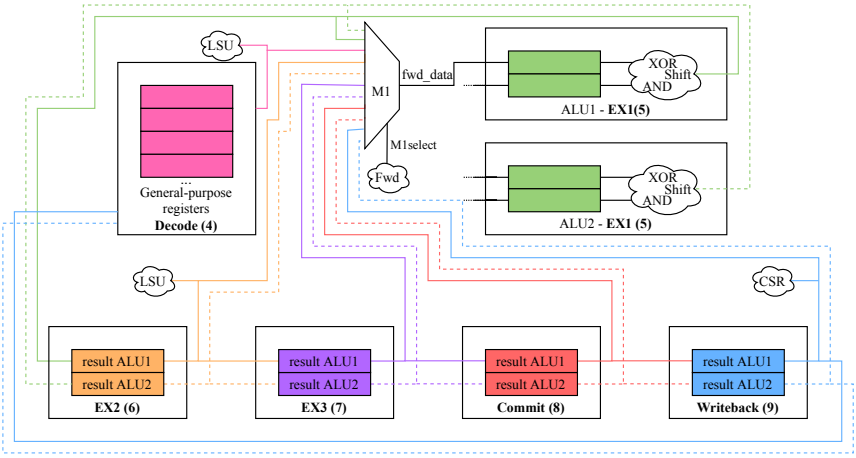


Figure 3.: Pipeline stages 4-9 in SweRV. Shares reside in the register file (■), are then sent to the ALU (■) before being buffered in pipeline registers (■, ■, ■, ■). Forwarding values from the pipeline registers to the ALU is possible in each stage and handled by the multiplexer M1, and the respective select signal M1select.

was only just calculated by the previous instruction. In the context of masking, this architectural design causes problems in two different points.

Figure 3 shows a simplified depiction of SweRV's pipeline stages 4-9. The multiplexer M1 is responsible for selecting which data is used as input for EX1, the first of the execution stages. This data either comes from the register file (GPR), the (LSU), or from any of the later execution stages due to forwarding logic. The select signal of this multiplexer, M1select, is computed in the respective pipeline stage from combinatorial logic and is therefore susceptible for glitches. Consequently, an attacker probing the output of M1, `fwd_data`, could, in the worst case, observe all of M1's possible inputs within one clock cycle until the select signal stabilizes. This means that if multiple shares of the same native value are in different pipeline registers, a combination of those can be observed at `fwd_data` in the same clock cycle. On top of that, since two different instructions are executed by SweRV at the same time, `fwd_data` can also combine data from the other execution unit. Exactly this problem was also seen in our empirical evaluation in Section 3.4.

In software, special care is also needed for control transfer instructions like conditional jumps. The instructions `beq`, `bne`, `blt` and `bge` perform conditional branches on data but are typically not used in (symmetric) cryptographic implementations to avoid potential timing side channels. Still, they can be used together with the unconditional jump instructions `jal` and `jalr` to implement

loops or function calls. In the context of masking, these instructions can cause problems which are invisible via e.g. the control flow graph of the software. Since the SweRV core decodes two instructions per cycle, the jump is potentially decoded with the instruction which comes code-wise after it. If this instruction operates on shares, and there are still shares of the same native value in the pipeline, a leak occurs, before the CPU realizes a change in the instruction pointer caused by the branch two cycles later. The instructions in these two cycles must be unrelated instructions, which requires in total four unrelated instructions.

Possible Hardware Solutions One could first consider to solve this problem in hardware by using a trick similar to the one used to prevent unintended glitches in the multiplexer tree of the register file. For example, one could gate the output of each pipeline register with a bit indicating whether the respective value should be forwarded back to the first execution stage (5) or not. This would further require individual gate-bits to be glitch-free, i.e., to be computed in the previous clock cycle and buffered in a register. The problem with pre-computing gate-bits is that those values are typically only available in the same cycle like the forwarding signal. One can overcome this problem by introducing additional pipeline stages in between the execution stage, however, this would significantly impact the overall performance of the core, also in cases where ordinary non-masked software is executed. Since we do not consider such a performance degradation to be a viable option, we next explore if those problems can better be dealt with on software-level.

Possible Software Solutions For a masked software implementation to not be affected by the side-effects of SweRV's forwarding logic, it must ensure that at no time there are two or more shares corresponding to the same native value in any execution stage of either execution unit. For example, if we consider the execution of two instructions, each of which uses a different share of the same native value, then one would need to ensure that there are $2 \times 6 + 1$ unrelated instructions between them. Hereby, an amount of 6 instructions is needed to clear all execution stages (5-9) of one execution unit, that then has to be doubled since SweRV has two execution units in total. Unrelated instructions are instructions processing data unrelated to any share (for example a `nop`), or shares from another native value.

While such a software constraint can significantly decrease the performance of masked software implementations, it is a solution that does not impact the performance of ordinary non-masked software. Nevertheless, as we will show later in Section 6, it is still possible to implement efficient masked software implementations fulfilling this constraint if the right masking/implementation techniques are used.

Software Constraints for ALU Operations

- *(Pipeline Stages and Execution Units)* Two instructions using different shares of the same native value must be separated by $6 \times 2 + 1$ unrelated instructions.
  *Combination of up to **13** shares possible (**big** problem).*

- *(Control transfer instructions)* Control transfer instructions, which are preceded by instructions processing shares, must be followed by 4 unrelated instructions.
  *Combination of up to **4** shares possible (**big** problem).*

4.2. Management Components of Data Memory

The SweRV core manages communication to the data memory via the Load-Store Unit (LSU). The LSU is a component between the CPU and the memory to ensure low memory latency by providing buffers and a dedicated pipeline for store operations. Our analysis shows, that the LSU Bus Buffer, responsible for saving values of recent loads or stores, similar to a data cache, turns out to be a major source of leakage which potentially combines multiple shares (***big*** problem). Furthermore, the dedicated store pipeline, components in the data memory interface, back-to-back memory accesses and the dedicated register file write port for memory accesses potentially combine two shares (***small*** problems). For each of these problems, we discuss possible solutions in hardware and software.

LSU Bus Buffer

Since data memory is connected to the SweRV core over an AXI4 bus, which can potentially introduce a considerable amount of latency, the LSU implements the so-called the LSU Bus Buffer, which works in principle like a small data cache. The LSU Bus Buffer consists of eight elements that are used to temporarily store the values of recent load or store events. Each element additionally stores the target address, an age, and a state, since the LSU uses a state machine to manage the buffer entries. Initially, all element states are set to *Idle*, meaning that they are ready to receive data, and their age is set to 0. While executing the memory access, the state and age are updated accordingly, until the memory access is finished and the element enters the *Idle* state again. However, the element is not removed from the buffer until the buffer is full and the oldest element is overwritten.

In the context of masked software implementations two problems arise in the LSU Bus Buffer. First, it is problematic if one share of a native value in the buffer and is overwritten with its counterpart, which might happen, e.g., when loading two shares from the data memory. This is not only a problem for load operations within short succession but can also occur if these operations are far apart since buffer elements are not cleared once their state goes back to *Idle*.

```
1  # Reset state/age of buffer elements
2  fence
3  # Load share 1 from address 0x20
4  lw x1, 0x20
5  # Reset state/age of buffer elements
6  fence
7  # Dummy overwrite of buffer element 1
8  lw x0, (x0)
9  # Reset state/age of buffer elements
10 fence
11 # Load share 2 from address 0x40
12 lw x2, 0x40
```

Figure 4.: Example of flushing the LSU buffer to clear it from shares

Second, if multiple shares of the same native value are stored in the buffer at the same time but at different locations, one can observe similar side-effects as originally described for ordinary register files (c.f. Section 3.1). The second problem could in principle be solved in hardware by applying a similar gating mechanism as for the the register file. However, in case of the LSU buffer, such a solution requires an additional register layer for pre-computing stable one-hot encoded signals, which decreases the performance of all software.

Instead, we can solve this problem on software-level by ensuring that the buffer holds at most one share per native value, which additionally prevents the problem of overwriting shares. When doing so, we could ideally target individual elements of the buffer such that a share can easily be overwritten with dummy data whenever needed. However, the LSU buffer is completely invisible from a progammer's perspective, which is also why there is no easy way to manipulate specific elements from software side. The choice of which buffer element is overwritten is determined by the element age, which depends not only on time, but also instruction dependencies, data addresses and the element state. While one could formulate a software constraint that takes all of these factors into consideration, we do not consider this a worthwhile solution due to complexity. Instead, we propose a software solution utilizing RISC-V's `fence`, that while introducing a 2-5 cycle overhead for each usage, is significantly easier to use in a correct way. In general, a `fence` can be used to ensure a certain order of memory operations by stalling the CPU pipeline unless previous load/store operations are finished. For the LSU Bus Buffer this means that all buffer elements are set to *Idle* with age 0. However, the stored values are not cleared, which must be done manually by executing load/store instructions dealing with unrelated data. The `fence` ensures that these loads and stores are inserted consecutively into the buffer, i.e., starting at the first slot and ending at the last slot, finally overwriting all buffer elements. It is recommended to place a second `fence` after this load/store sequence, before loading or storing further shares. Figure 4 shows a short exemplary code snipped, in which a share is stored in the buffer and later cleared.

Store Pipeline Stages

Before being stored to memory, data values pass through three dedicated pipeline stages in the LSU (c.f. Figure 1), which are exclusively updated when a store happens. A share used as an operand in a store instruction will therefore hang in the pipeline until it is overwritten by the data of the next store. This is problematic if the data of the next store is a share from the same native value. In order to avoid this problem, it is sufficient to ensure that at least one unrelated store operation is performed between two stores that transfer two shares of the same native value.

Data Memory Interface

SweRV reads 8 bytes from the external data memory module in one cycle, and then selects the parts which are effectively needed according to the load address and load instruction (`lw`, `lh` or `lb`). Glitches in the selection signal can lead to problems if two shares of a native value are stored in the same 8-byte data memory word. A hardware gating mechanism is again not viable since it would increase latency, which is why we suggest to store shares of the same native value not within the same 8-byte word.

Back-to-back Memory Accesses

SweRV is able to execute memory accesses in a back-to-back fashion, i.e., in two consecutive cycles. Given a data memory layout that utilizes partially one-hot encoded addresses (c.f. Section 3.1), an additional problem can occur if shares s_i, s_j are stored stored in block $b1$ at indices i, j respectively, and one accesses unrelated data at indices i, j in another block $b2$ in two consecutive cycles. The the output of $b1$, even though ignored, will switch from s_i to s_j in two consecutive cycles. Preventing this kind of leakage can be done by paying special attention to the block indices during each memory access, or reserving one "neutral" index within blocks that never holds any shares and thus can be used for inserting a dummy load.

Register file gating for Data from Memory

Register file write ports of the SweRV core need to be gated by a stable gate bit (c.f Section 3.1). Computing the gate bit is straightforward for all write ports except for the one dedicated to data loaded from memory, since it depends on a potentially glitching write enable signal derived from the LSU bus buffer entries. First, we gate the write data with the stable register write address only, which means the preliminary gate bit is set for all registers which have loads pending in the LSU bus buffer. In the next cycle, the write enable value is then used to select the final, correct write register. This solution requires the software to

ensure that no other pending load in the LSU bus buffer writes to a register, which contains another share from the same native value.

Software Constraints for Memory Operations

- *(LSU Bus Buffer)* Two memory accesses processing two shares must be separated by a `fence`, followed by a load of unrelated data, followed by a `fence`.
  *Combination of up to **8** shares possible (**big** problem).*
- *(Store Pipeline Stages)* Two stores storing two shares must be separated by a `fence`, followed by a store of unrelated data, followed by a `fence`.
  *Combination of up to **2** shares possible (**small** problem).*
- *(Data Memory Interface)* Shares must be stored in the same memory block, but not within an 8-byte word.
  *Combination of up to **2** shares possible (**small** problem).*
- *(Back-to-back memory accesses)* Either one 8-byte region per block at index i is not used to store shares and between any two loads, a load to this region is performed, or if a share s_i is stored at index i and s_j is stored at index j in a block, no back-to-back accesses to any addresses mapping to index i and j are performed.
  *Combination of up to **2** shares possible (**small** problem).*
- *(Write port 2 of the register file)* If a share s_i is stored in register x_i and s_j is stored in memory, then there must not exist another load at the same time which writes to register x_i.
  *Combination of up to **2** shares possible (**small** problem).*

5. Deriving Generic Software Rules

In this section, we propose generic rules for the design of masked software implementations that are intended to run on more complex CPUs like SweRV. These rules take into account features like pipeline length, the number of execution units, and the size of load/store buffers, and are based on the software constraints defined in Section 4. We also discuss the *lazy engineering* approach by Balasch et al. [Bal+14] and demonstrate that, while entirely relying on this approach in our setting is not recommended, it can still be a worthwhile trade-off that can eliminate many smaller problems, that would otherwise all need be dealt with in software.

5.1. Generic Rules for Masked Software

A CPU can be described by numerous characteristics, ranging from the architecture width to register file size to cache sizes. Our analysis in Section 4 shows

that, when considering the implementation of masked software implementations, the following characteristics are especially important:

- The amount of pipeline stages p
- The amount of execution units e
- The size of data buffers.

Pipelines and Execution Units Forwarding logic, also known as bypass-logic, is a common optimization in pipelined CPUs which we identify to be a big problem for masked software implementations. In the worst case, each pipeline stage forwards its current content to the first stage, where it can be effectively combined with data from all stages due to glitches. Assuming a pipeline length $p = p_i + p_d$, where p_i is the number instruction fetch stages and p_d is the number of decode/execute stages (processing actual data), this problem can be avoided by ensuring that at least $p_d + 1$ unrelated instructions are executed between any two instructions processing the shares of the same native value. We have observed this problem on the SweRV core ($p_i = 3$, $p_d = 6$) but it also affects simpler cores such as the CV32E40P (formerly known as RI5CY) that is roughly comparable to an ARM Cortex M4 [Ope]. This core features a 4-stage pipeline ($p_i = 1$, $p_d = 3$), and would therefore still require a "padding" with four unrelated instructions.

On top of that, more powerful CPUs like SweRV often feature a superscalar architecture, including e.g. a dual-issue pipeline, that allows executing two instructions per clock cycle. This is achieved by having e execution units in parallel, all of which have their own fetch/decode/execute stages. In those cases, forwarding is not only possible between stages of the same execution unit but also across them. This additionally increases the required amount of padding to $e \times p_d + 1$.

Data Buffers and Caches Besides pipeline stages, another big problem for masked software implementations is the existence of data buffers that are invisible from a programmers perspective. Defining generic rules for these components is somewhat harder as their exact behavior can differ quite a lot depending on their concrete implementation. However, typically when considering SweRV's LSU buffer or many other cache designs, these components can cause shares to essentially get stuck at certain locations within the CPU where they then represent an additional source of leakage from this time onward. While such problems can be resolved in hardware, e.g. as shown for the register file (c.f. Section 3.1), this is only really a viable option in cases where these hardware modifications do not increase latency, which is also why we need to deal with SweRV's LSU buffer side effects in software. In general one needs to ensure that, whenever a share is transferred over an unmodified buffer, none of the other buffer entries contain shares that correspond to the same native value. How this

can be achieved is implementation dependent. In the easier case, a mechanism to clear the buffer contents could be implemented in hardware, which is however not always efficient since it would also affect unmasked data. In the harder case, one has to make use of dummy loads/stores to clear all unwanted values.

Rules Here we summarize the most important rules for masked software on application-level processors. As we explain in Section 5.2, many of the other smaller problems are probably better dealt with using the "lazy engineering" approach.

`R1` Two instructions processing shares from the same native value must be separated by $e \times p_d + 1$ unrelated instructions.

`R2` Whenever a share is transferring through a buffer, none of the other buffer entries must contain shares that correspond to the same native value.

Naturally, at this point one could also ask how these rules would look like on even more complex CPUs with multi-level caches, out-of-order execution, or speculative memory accesses. For example, the 64-bit out-of-order RISC-V BOOM core would be a potential target for further analysis. However, when considering the analysis of such CPUs we currently see quite a few problems that are not necessarily easy to overcome. First, out-of-order execution will violate our assumption of having software with constant control flow, meaning that verifying a program's execution once might not be indicative of future runs. Second, the effects of, e.g., large cache hierarchies will likely cause problems where corresponding software constraints would become too complex to implement with reasonable effort and overhead. Nevertheless, we argue that physical attacks like power analysis are most relevant only for devices in the range from microprocessors to application processors. An attacker having physical access to a desktop/server could anyway use other methods, like cold boot attacks, to compromise a system more efficiently.

5.2. The Cost of Lazily Engineering

Until now, the verification of masked software implementations is mostly done using rather simple security models like the value-based or register-based leakage model. While such models are certainly useful to detect some problems, many other works also show that processors do emit leakage that is not captured by these models [Bal+14; Gao+20; Gro+16; MMT20; PV17]. Balasch et al. [Bal+14] formalize this behaviour in their order-reduction theorem, which states that on simple CPUs, the security of dth-order masked software in the value-based leakage model reduces to $\lfloor \frac{d}{2} \rfloor$-th order in the transition-based leakage model. In other words, as a rule-of-thumb for a "lazy" software engineer, they suggest to double the security-order of a masked implementation to achieve the desired security-order in a model that more accurately reflects reality.

While these works focus on rather simple microprocessors, our analysis has shown that on more complex application-level processors, the reduction of security order can be significantly higher. When deriving the expected security reduction of *lazy engineering* on application-level processors, the main point to consider is the component that can potentially combine the most shares. In the case of our modified SweRV this component would be the forwarding logic of the CPU pipeline. According to our generic rules, a processor executing algorithmically correct masked software, might combine up to $e \times p_d + 1$ shares in its pipeline. Consequently, without any further assumptions, the CPU could create all combinations of any choice of $e \times p_d + 1$ shares, which corresponds to an order reduction of $\left\lfloor \frac{d}{e \times p_d + 1} \right\rfloor$.

To give a concrete example, when relying entirely on lazy engineering, one would in theory require at least a 13th-order masked implementation for actual 1st-order security on SweRV in the time-constrained probing model. While we do not expect an easily exploitable order reduction this large when performing physical power measurements of SweRV (c.f. Section 3.4), we also want to stress that these architectural side effects should not be underestimated. For example, works like [MMT20] show that, already on simple microprocessors, a generic 2nd-order masked software implementations can very well loose both of its protection orders in practice. If we add to that the fact that SweRV's architecture has the potential to unintentionally combine many more shares at many more locations, one can expect that quite a few masking orders will also be required in case of SweRV. Given that masking imposes a runtime overhead that is quadratic in the masking order, such very high-order implementations might however still not be a desirable solution, especially in automotive applications with real-time requirements. As we show later, in such cases, we recommend utilizing lazy engineering only for eliminating small problems while tackling big problems using more effective implementation/masking strategies that we describe in Section 6.

6. Evaluation

In this section we demonstrate that, despite the fact that cores like SweRV can cause significant problems for masked software implementations in general, one can still design fine-tuned, secure versions with very small overhead. First, we explain how one can use a *parallel* instead of the usual *serial* coding strategy to significantly reduce the performance impact of software constraints that require a separation between processing shares of the same native value. We then explain how one can utilize Threshold masking schemes, and by extension also the core idea of lazy engineering, to design masked software for SweRV that is secure, efficient, and easy to implement. More concretely, we show that the runtime overhead of e.g. a masked Keccak S-box implementation providing 1st-order security on SweRV, when compared to a corresponding implementation ignoring all software constraints, can be as little as 13%.

Table 1.: Runtime comparison of masked software implementations on the SweRV core. Plain implementations do not consider software constraints, and thus lose all protection orders. Secure implementations are handcrafted for SweRV, consider all required constraints, and can thus preserve their claimed protection order. NOPs indicate the required amount of `nop`'s or dummy loads/stores. Testcases marked with *reg.* do not perform any memory accesses, i.e., all data is in the register file at the beginning/end of the computation.

			Plain Implementations		Secure Implementations			
Name	Input Shares	Fresh Randomness	Cycles	Instr-uctions	Cycles	Instr-uctions	NOPs	Verification Runtime
Tornado-generated Implementations								
ISW Keccak S-box	10 × 32 bit	5 × 32 bit	163	330	-			
ISW Keccak S-box, 2nd order	15 × 32 bit	15 × 32 bit	1272	810	-			
ISW Keccak S-box, 3rd order	20 × 32 bit	30 × 32 bit	2124	1121	-			
ISW Keccak S-box, 4th order	25 × 32 bit	50 × 32 bit	4406	3309	-			
AND Gate Implementations								
DOM AND *reg.* [GMK16]	4 × 32 bit	32 bit	10	8	33	48	40	1.4 m
ISW AND *reg.* [ISW03]	4 × 32 bit	32 bit	10	8	32	48	40	57 s
TI AND *reg.* [NRR06]	4 × 32 bit	-	14	15	37	54	39	1.1 m
Trichina AND *reg.* [Tri03]	4 × 32 bit	32 bit	9	8	34	46	38	1.28 m
DOM AND *reg.*, 2nd order [GMK16]	6 × 32 bit	3 × 32 bit	20	21	86	148	127	3.2 m
DOM AND *reg.*, 3rd order [GMK16]	8 × 32 bit	6 × 32 bit	33	42	250	295	235	9.6 m
Serial/Parallel Implementations								
DOM Keccak S-box *reg.*, serial [GSM17]	10 × 32 bit	5 × 32 bit	83	95	240	418	333	8.4 m
DOM Keccak S-box *reg.*, parallel	10 × 32 bit	5 × 32 bit	36	60	81	144	79	3.7 m
DOM Keccak S-box, serial [GSM17]	10 × 32 bit	5 × 32 bit	174	140	550	624	464	22.38 m
DOM Keccak S-box, 2nd order, serial	15 × 32 bit	15 × 32 bit	283	250	2050	1465	283	1.5 h
Threshold Implementations								
TI Keccak S-box, *reg.*	15 × 32 bit	-	66	105	72	126	15	3.5 m
TI Ascon (1 round)	15 × 64 bit	-	721	863	1621	1153	290	1.18 h

Evaluation Setup All of our tested implementations are hand-written assembly code, except for the *Tornado*-generated C implementations that are compiled with the compiler flag `-O1`. For the verification and performance benchmarks we used the cycle accurate simulation of SweRV's netlist within Coco. Coco itself was executed on a 64-bit Linux operating system on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. We configure SweRV with data memory ranging from 256 byte to 2 KB, adapted as required by the respective testcase. The instruction memory and instruction cache is configured to be 2 KB for each test case.

The SweRV configuration using 256 byte of data memory, after applying the optimizations described in Section 3.2, results in a circuit with 420 000 gates, of which 108 000 are registers and 97 000 are non-linear gates. A detailed breakdown of these numbers can be found in Appendix B. This makes the hardware design of SweRV orders of magnitudes larger than the Ibex design which was studied in [Gig+21], and consisted of only 27 000 gates.

Software Implementation Package To measure the overhead imposed by different software constraints, we construct a comprehensive set of masked software implementations. First, we take a look at several examples of masked And gates, which represent the simplest non-linear function (degree 2). More concretely, we analyze 1st-order implementations of the Ishai-Sahai-Wagner (ISW) And [ISW03], the Trichina And [Tri03], the Threshold Implementation (TI) And [NRR06], and up to 3rd-order masked variants of the Domain Oriented Masking (DOM) And [GMK16].

We then investigate masked S-box implementations which represent the non-linear layer within symmetric cryptographic computations, and use masked And-gates as basic building blocks. Here, we focus on 1st- and 2nd-order masked implementations of the Keccak S-box, which has a prominent use in the SHA-3 hash function. Furthermore, we provide TI variants of the Keccak S-box [Ber+11], as well as one complete round (linear + non-linear layer) of the Ascon cipher [Dob+16].

In Table 1 we list *plain* implementations, which are correct in the value-based leakage model, but do not consider any of SweRV's software constraints, and are thus also not secure on this core when verified by Coco in the time-constrained probing model. In contrast, *secure* implementations fulfill all required constraints can thus be verified successfully for their claimed protection order on SweRV. For each implementation, we report SweRV's execution runtime in cycles, as well as the number of executed instructions. Additionally, for secure implementations, we report the number of unrelated instructions (NOPs), that are needed to achieve the required amount of time separation between the processing of shares of the same native value, as stated by the individual software constraints.

6.1. Serial vs. Parallel Implementations

Many modern symmetric cryptographic primitives have a mathematical description based on simple Boolean functions that can be easily mapped into a corresponding software/hardware implementation. For example, the Keccak S-box (as used in SHA-3) operates on a state consisting of five *lanes*, each of which is combined with two other lanes using a sequence of simple And, Xor, and Not operations to compute the corresponding output lane. The most straightforward way of implementing this S-box in software is to take a set of three lanes, processing them, storing the resulting output lane, and repeating these steps until the computation of all five output lanes is finished.

If we now consider a masked implementation, where each input/output lane is represented by two (or more) shares, the same implementation strategy can be used, except for the fact that the sequence of Boolean operations needs to be adapted such that (1) shares of the same native value (lane) are never directly combined, (2) the (native) output is still the same. If we further consider a software constraint that requires a certain amount of unrelated instructions between processing shares of the same native value, one can imagine that additional `nop` instructions will need to be introduced for this purpose. Alternatively, one could consider a *parallel* implementation, where one interleaves the computation of the five output lanes such that `nop`'s can be replaced with computations on shares of other lanes. We give an example that illustrates the runtime difference between serial and parallel implementations in Appendix A. This runtime difference is also quite visible in Table 1. For example, the parallel DOM Keccak S-box implementation (81 cycles) is three times faster than its serial counterpart (240 cycles).

One potential downside of parallel implementations is the fact that they increase the maximum amount of intermediate values that need to be kept track of. Especially in case of higher-order masked implementations, a processor's register file might not be large enough to hold this increased amount of intermediate values. The resulting register spilling then requires additional load/store operations that also need to comply with software constraints and can thus eliminate any potential gain of this approach. To illustrate the overhead of memory operations, we have included a serial implementation of the Keccak S-box that initially loads all shares from memory and computes the S-box. If we compare the runtime of this implementation (550 cycles) to the serial implementation that performs computations without intial memory operations (240 cycles), we can observe a runtime overhead of about factor two.

6.2. Threshold Implementations

Threshold implementations (TI) [NRR06], is a provable secure masking scheme that splits non-linear functions into multiple incomplete *component functions*. More concretely, in TI, each component function fulfills the non-completeness

property, meaning that its computation is independent of at least one of its input shares. One consequence of incompleteness is that TI schemes require computations with at least three shares in order to provide 1st-order security. At the same time, this incompleteness guarantees that any combination of intermediate values during the computation of one component function can combine at most two out of three shares of any native value, therefore leaving 1st-order security intact.

In the context of implementing secure and efficient masked software implementations for SweRV, TI turns out to be beneficial in two ways. First, the "lazy" characteristic of TI allows us to ignore all ***small*** problems that can combine at most two shares. Second, a TI description of Keccak, for example as shown in [Bil+13], also gives a description of three S-box component functions, each of which only contain instructions that operate on an incomplete set of shares. Hence, when implementing TI Keccak in software, one can calculate the linear layer in sequence for each share, and the non-linear layer in sequence for each component function. Then, one only really needs to pay attention to ***big*** problems when switching the calculation from one component functions to another. This significantly simplifies the software development process as ***big*** problems can only really occur twice per Sbox computation.

In Table 1, we show a TI implementation of the Keccak S-box (72 cycles) which has almost no overhead compared to the corresponding plain implementation (66 cycles). Compared to a plain parallel DOM implementation, the overhead of a secure TI implementation is still only a about a factor of two, while being at lot easier to implement. With TI Ascon, we also present runtimes of implementations that compute an entire cipher round (linear + non-linear layer). The choice of using Ascon for this comparison is motivated by the fact that Ascon uses a S-box very similar to Keccak, and a linear layer that is significantly easier to implement in assembly than in case of Keccak. From the reported numbers we can see that only 290 additional `nops` are needed to make this implementation conform to the required software constraints. While the cycle count of the secure implementation is still about twice as large as in the plain case, we want to stress that most of this overhead ($\approx$ 900 cycles) is due to software constraints for data memory since three shares of Ascon's state do not quite fit into the register file anymore.

7. Conclusion

In this work, we have performed a comprehensive analysis of more complex CPU architectures in the context of masking-related side effects. First, we showed that on cores like SweRV, there exists a significant gap between security in a simple software probing model and practical security for masked software. We underlined this point both via a formal analysis in the hardware probing model and via empirical analysis based on gate-level timing simulations. We then further analyzed the components of SweRV in the hardware probing model,

identified new problems, and discuss possible solutions in terms of software constraints. Ultimately, while there exist many hardware components that can reduce the security of masked software due to architectural side-effects, we show that there only exist a few components that could reduce the security of masking schemes by multiple orders. Hence, when considering the implementation of efficient masked software for such CPUs, we recommend to use a combination of TI/lazy engineering to deal with small problems while only addressing the few large problems directly in the software implementation. In that case, the performance overhead of software constraints can be as low as 13% while the resulting implementation can be fully formally verified on our *secured* SweRV in the hardware probing model. If 2nd-order protection is desired, one could again rely on TI/lazy engineering for small problems, here however the additional cost of this approach might not justify this convenience anymore. When aiming for even higher protection orders, one likely needs to consider all software constraints directly in the implementation to keep the runtime overhead manageable.

Acknowledgements

This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", and the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, and via the FERMION project (grant nr 867542).

Appendix A.

```
# Shares lane 0: x2,
#     x3
# Shares lane 1: x4,
#     x5
# ...
# Randomness: x12,
#     x13, x14,
#     x15, x1
# Lane 0
not x17, x2
and x24, x17, x5
xor x24, x24, x12
and x25, x3, x4
xor x25, x25, x12
and x27, x17, x4
xor x27, x27, x24
and x28, x3, x5
xor x28, x28, x25
xor x27, x27, x10
xor x28, x28, x11
# Lane 1
not x17, x4
and x24, x17, x7
xor x24, x24, x13
and x25, x5, x6
...
#Lane 2
...
```

```
# Shares lane 0: x2,
#     x3
# Shares lane 1: x4,
#     x5
# ...
# Randomness: x12,
#     x13, x14,
#     x15, x16
  # NOT
  not x17, x2
  not x18, x4
  not x19, x6
  not x20, x8
  not x21, x10
  #DOM-AND - Instr 1
  and x22, x17, x5
  and x23, x18, x7
  and x24, x19, x9
  and x25, x20, x11
  and x26, x21, x3
  #DOM-AND - Instr 2
  xor x22, x22, x12
  xor x23, x23, x13
  xor x24, x24, x14
  xor x25, x25, x15
  xor x26, x26, x16
  #DOM-AND - Instr 3
  ...
```

Figure 5.: Comparison between serial and parallel DOM Keccak S-box

Appendix B.

Table 2.: Circuit size of the SweRV core (256 byte of data memory, 2 KB of instruction memory / cache) before and after optimization (Removal of unused instruction memory logic)

	Raw circuit	Optimized circuit
Registers	108 129	108 043
Linear Gates	8 828	8 708
Non-linear Gates	133 415	97 222
Not-Gates	3 518	3 248
Multiplexers	335 294	203 107
Total	**589 188**	**420 332**

References

[Ath+20] Konstantinos Athanasiou, Thomas Wahl, A. Adam Ding, and Yunsi Fei. "Automatic Detection and Repair of Transition-Based Leakage in Software Binaries". In: *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Ed. by Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel. Vol. 12549. Lecture Notes in Computer Science. Springer, 2020, pp. 50–67.

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*. Vol. 8968. Lecture Notes in Computer Science. Springer, 2014, pp. 64–81.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 457–485.

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 116–129.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[Bar+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 189–228.

[Bay+13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. "Sleuth: Automated Verification of Software Power Analysis Countermeasures". In: *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Springer, 2013, pp. 293–310.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 397–426.

[Bel+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In: *EUROCRYPT (3)*. Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–341.

[Ber+11] Giulio Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak Reference*. 2011.

[Bil+13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. "Efficient and First-Order DPA Resistant Implementations of Keccak". In: *CARDIS*. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013, pp. 187–199.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic*

Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 69–81.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2. Submission to the CEASAR Competition*. https://ascon.iaik.tugraz.at/files/asconv12.pd. Retrieved on February 4th, 2021. 2016.

[EWS14] Hassan Eldib, Chao Wang, and Patrick Schaumont. "Formal Verification of Software Countermeasures against Side-Channel Attacks". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (2014), 11:1–11:24.

[Fau+10] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. "Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases". In: *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 135–156.

[Gao+20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. "Share-slicing: Friend or Foe?" In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 152–174.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Gro+16] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. "Bitsliced Masking and ARM: Friends or Foes?" In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*. Vol. 10098. Lecture Notes in Computer Science. Springer, 2016, pp. 91–109.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 2017, pp. 205–212.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification". In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1297.

[Mos+12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. “Compiler Assisted Masking”. In: *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 58–75.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. “Threshold Implementations Against Side-Channel Attacks and Glitches”. In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[Ope] OpenHW Group. *OpenHW Group CV32E40P User Manual: Pipeline Details*. https://cv32e40p.readthedocs.io/en/latest/pipeline/, Retrieved on January 26th, 2021.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. “Mind the Gap: Towards Secure 1st-Order Masking in Software”. In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 282–297.

[Ren+11] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. “A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices”. In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 109–128.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. “Consolidating Masking Schemes”. In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[RP10] Matthieu Rivain and Emmanuel Prouff. “Provably Secure Higher-Order Masking of AES”. In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427.

[Sny22] Wilson Snyder. *Verilator*. https://www.veripool.org/wiki/verilator. Retrieved on February 2nd, 2021. 2022.

[Ted19] Ted Marena, Western Digital. *The Journey of RISC-V Implementation*. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/article-journey-of-RISC-V-implementation.pdf, Retrieved on January 16th, 2021. 2019.

[The] The Regents of the University of California. *RISCV-BOOM: The Load/Store Unit (LSU)*. https://docs.boom-core.org/en/latest/sections/load-store-unit.html, Retrieved on January 27th, 2021.

[Tri03] Elena Trichina. "Combinational Logic Design for AES SubByte Transformation on Masked Data". In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 236.

[Wesa] Western Digital. *RISC-V SweRV EH1 Programer's Reference Manual*. https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf, Retrieved on January 16th, 2021.

[Wesb] Western Digital. *RISC-V: high performance embedded SweRV core microarchitecture, performance and CHIPS Alliance*. https://riscv.org/wp-content/uploads/2019/04/RISC-V_SweRV_Roadshow-.pdf, Retrieved on January 16th, 2021.

[Wes19] Western Digital. *RISC-V and Open Source Hardware Address New Compute Requirements*. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/tech-brief/tech-brief-western-digital-risc-v.pdf, Retrieved on January 16th, 2021. 2019.

[Wol16] Claire Wolf. *Yosys Open Synthesis Suite*. http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021. 2016.

[WSW19] Jingbo Wang, Chungha Sung, and Chao Wang. "Mitigating power side channels during compilation". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 590–601.

[Zha+18] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. "SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 157–177.

5

Secure Context Switching of Masked Software Implementations

Publication Data. Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure Context Switching of Masked Software Implementations".
In: *AsiaCCS*. 2023.

Contribution. The author of this thesis proposed the concepts, performed all the experiments, made the implementations and wrote most of the text.

Secure Context Switching of Masked Software Implementations

Barbara Gigerl[1], Robert Primas[1], Stefan Mangard[1]

[1] Graz University of Technology

Abstract Cryptographic software running on embedded devices requires protection against physical side-channel attacks such as power analysis. Masking is a widely deployed countermeasure against these attacks and is directly implemented on algorithmic level. Many works study the security of masked cryptographic software on CPUs, pointing out potential problems on algorithmic/microarchitecture-level, as well as corresponding solutions, and even show masked software can be implemented efficiently and with strong (formal) security guarantees. However, these works also make the implicit assumption that software is executed directly on the CPU without any abstraction layers in-between, i.e., they focus exclusively on the bare-metal case. Many practical applications, including IoT and automotive/industrial environments, require multitasking embedded OSs on which masked software runs as one out of many concurrent tasks. For such applications, the potential impact of events like context switches on the secure execution of masked software has not been studied so far at all.

In this paper, we provide the first security analysis of masked cryptographic software spanning all three layers (SW, OS, CPU). First, we apply a formal verification approach to identify leaks within the execution of masked software that are caused by the embedded OS itself, rather than on algorithmic or microarchitecture level. After showing that these leaks are primarily caused by context switching, we propose several different strategies to harden a context switching routine against such leakage, ultimately allowing masked software from previous works to remain secure when being executed on embedded OSs. Finally, we present a case study focusing on FreeRTOS, a popular embedded OS for embedded devices, running on a RISC-V core, allowing us to evaluate the practicality and ease of integration of each strategy.

1. Introduction

Embedded devices have become omnipresent in IoT, automotive, and industrial applications and often interact with their physical environment. This raises the need for strong cryptographic primitives to preserve private and secure operations. Embedded devices need to be protected against both theoretical and physical attacks. Theoretical security refers to guarantees such as the resistance of cryptography against mathematical attacks, while physical security counteracts

adversaries in physical proximity who observe a device's physical properties during computation. In 1999, Kocher et al. [KJJ99] presented Differential Power Analysis (DPA), which allows for extracting secrets like cryptographic keys from a device. DPA is performed by observing a device's power consumption, which correlates with the processed data. Since then, masking has become a very popular and well-studied countermeasure to defeat such attacks on algorithmic level [Bar+17; Bel+17; Cha+99; Cnu+16; GM17; GP99; ISW03; Rep+15]. With masking, each secret variable of a cryptographic computation, such as the encryption key, is split into $d + 1$ random shares. Consequently, the power consumption of the device does not correlate with the unshared secret but with the $d + 1$ random shares, which exponentially increases the difficulty of recovering the unshared secret. One particular advantage of masking is that it is provably secure, i.e., it can be proven that an attacker cannot reveal any information about the unshared secret by combining up to d shares.

In the past, many works have pointed out a significant gap between the theoretical and practical security of masked implementations [Bal+12; Gig+21; PV17], often caused by physical effects such as glitches and transitions. Masking schemes generally assume that independent computations result in independent leakage, which is not necessarily the case in a practical software or hardware implementation. In other words, a masked software implementation that has been formally proven to be d-th order secure in theory might not reach this security level when executed on a CPU. Many works in the past have discussed to which extent the CPU microarchitecture can compromise the security of masked software implementations. Prominent root causes of order-reducing leakage in masking are register or memory overwrites, which leak the Hamming distance between two shares [Bal+14; BDV21; Cor+12; PV17]. On top of that, many more such potential problems have been identified that essentially boil down to implementation specifics of the register file, SRAM, load-store logic, data caches, or bypass mechanisms in the CPU pipeline [Gig+21; GPM21; Gro+16; MMT20]. In order to solve these problems efficiently, works like [Gig+21; GPM21] emphasize that modifications on software level are necessary while additional hardware changes of the CPU are advisable. Eventually, both the CPU hardware and the masked software implementation need to fit together to obtain secure execution that preserves the theoretical security of the masking.

In practice, with the exception of the most basic microcontrollers or IoT devices, embedded devices execute software within an (embedded) OS alongside other tasks which include bus or network communications, and sensor data acquisition and processing. In the case of resource-constrained embedded devices, one often chooses dedicated embedded OSs, including real-time operating systems (RTOS), over fully-fledged operating systems such as Linux. FreeRTOS [Ama22] is a very popular choice for such an embedded OS because it provides a wide range of supported platforms, a large community, and is publicly available (open-source).

So far, works on the practical security of masked software implementations

focus on the bare-metal case and therefore assume total control over the execution of software on a CPU [Bar+21; CGD18; Gig+21; GPM21; MOW17; MPW22; PV17; She+21a; She+21b]. More concretely, they assume that the masked software is not interrupted during execution. The interference of multitasking OSs, especially context switching, that leads to a violation of this assumption has not been considered in previous analysis efforts at all. Still, context switches occur at high frequencies, e.g., due to periodic (timer) interrupts, and in some cases, their occurrence can even be controlled by the attacker. Consider, for example, the following setting, in which an attacker first requests a certain cryptographic operation via a common communication interface and causes an IO interrupt at a later point in time by sending an additional request. The attacker can easily observe repeated executions of the cryptographic operation in which context switches cause additional leakage, allowing to easily mount straight-forward attacks like DPA on the additionally created leakage in the power side-channel [KJJ99].

As a countermeasure, one could consider the option of disabling interrupts during the execution of masked software; however, this option is unrealistic in practice due to (1) the starving of other relevant tasks like sensor data acquisition/processing or Bluetooth/UART/MQTT network communication, and (2) the generally strict scheduling requirements of RTOS systems that allows meeting timing constraints [AG21; BSH12; ZLG09]. There does exist one work by Balasch et al. [Bal+15] from 2015 demonstrating successful DPA attacks on an AES implementation executed by a Linux operating system on an ARM Cortex-A8, and discussing the security of masked software in this setting. The authors show that also the masked version of the AES is not leakage-free. However, it remains unclear whether the empirically found leakage is caused by the CPU microarchitecture, the OS, or even the masking algorithm itself. On top of that, they also do not propose a solution that can reliably prevent the observed leakage.

Contributions. The security of masked software implementations running as a task/process within an embedded OS has not been evaluated so far. It is hence unclear to what extent specific OS features like interrupts, scheduling, and context switches cause leakage in such situations and what corresponding protection mechanisms can be put into place at what cost. We close this gap by providing the first in-depth analysis of masked software executed by an OS on a CPU. The main contributions of this work are as follows:

- We provide the first formal analysis of masked software which runs as a task in an embedded OS on a CPU. Using a toy example, we show that the main problems are caused during context switching by either overwriting shares in memory, or transitions on memory/register file read/write ports (Section 4).

- We propose several possible strategies to solve these problems, resulting in

a formally verified context switching routine hardened against side-channel leakage that requires no assumptions on the current location of shares in the register file. This allows masked software, verified for correctness in the bare-metal case, to preserve security when executed on an embedded OS. For each strategy, we provide a comparison of their advantages, disadvantages, and performance overhead (Section 5).

- We present a case study of masked software running as a task in FreeRTOS on a RISC-V CPU. In this case study, we show that the problems identified in our analysis also exist in FreeRTOS and that these problems can be fixed efficiently by the proposed strategies. (Section 6).

- We make the evaluation setup and all software artifacts available on GitHub[1].

2. Background

In this section, we first give necessary background information on the masking countermeasure. We briefly introduce Coco, a formal verification tool to prove that the execution of (bare-metal) masked software on a given CPU netlist is secure. For our work, we use Coco as a leakage detection mechanism, as well as to formally verify the security of our countermeasures. Finally, we provide a short introduction to embedded operating systems.

2.1. Masking

Power analysis attacks exploit the fact that the power consumption of a cryptographic device depends on the processed data, such as a secret key [CRR02; KJJ99]. The masking countermeasure breaks this dependency by randomizing sensitive intermediate values processed by the device [Cha+99; GMK16; GP99; NRR06]. Each sensitive variable used in a cryptographic computation is split into $d + 1$ random shares, such that the observation of up to d shares does not reveal any information about the corresponding sensitive value.

In the case of a dth-order Boolean masking scheme, the shares $s_0 \dots s_d$ must satisfy $s = s_0 \oplus \dots \oplus s_d$, where $\oplus$ stands for the exclusive OR (XOR) operation. Hereby, $s_0 \dots s_{d-1}$ are chosen uniformly at random, while $s_d = s_0 \oplus \dots s_{d-1} \oplus s$, which ensures that each share s_i is uniformly distributed and statistically independent of s. For example, a first-order masking scheme ($d = 1$) splits up a sensitive variable s into two parts s_0 and s_1, such that $s = s_0 \oplus s_1$, s_0 is chosen uniformly at random, and $s_1 = s \oplus s_0$.

Throughout the entire implementation, a proper separation of shares and of the output of the component functions needs to be ensured not to violate the

[1] https://github.com/barbara-gigerl/sw-masking-rtos

d^{th}-order independence, which is commonly expressed in the probing model of Ishai et al. [ISW03]. In the probing model, the attacker has the ability to probe up to d intermediate results of the masked implementation. An implementation is said to be secure if the probing attacker cannot gain any statistical advantage in guessing any secret variable by combining the probed results in an arbitrary manner. While this share separation can be easily ensured for functions which are linear over $\mathrm{GF}(2^n)$ – for example, the masked calculation of $x \oplus y$ can be performed share-wise $(x_i \oplus y_i)$ – the secure implementation of nonlinear functions usually requires the introduction of fresh randomness.

The probing model can directly be applied to masked hardware circuits, in which the attacker can place probes on individual gates and wires, which then allow observing all values at the chosen location for an infinite amount of time. However, the probing model is less suitable for masked software implementations executed by a CPU. For example, the attacker could simply place one probe on the read or write port of the register file and then observe all intermediate values (including shares), which allows breaking masked software of arbitrary protection order. Instead, recent works refer to the time-constrained probing model [Gig+21] for masked software implementations, which puts a time restriction of one clock cycle on each probe. More formally, the attacker who possesses d probes can distribute these both spatially and temporally, allowing them to perform measurements at different locations in the same clock cycle, the same location in different clock cycles, or a mix of both. For example, a first-order attacker ($d = 1$) in the time-constrained probing model can only probe register file read or write ports for the duration of one clock cycle.

Besides algorithmic correctness of masking schemes in a respective probing model, the practical security of masked cryptographic algorithms also strongly depends on implementation specifics in hardware [AG01; Cnu+16; GMK16; GSM17; MMT20; NRR06] and software [Bal+14; Bel+20; Gro+16; Wan+15]. We discuss the case of secure software masking in more detail in the following.

2.2. Practical security of masked software

The security of masked software implementations depends on the assumption that independent computations result in independent leakage [Bal+14; Cha+99; GP99]. However, many works have shown that this property is often violated in practice when executing masked software (bare-metal) on CPUs [Bal+14; Gig+21; GPM21; MMT20; MPW22; PV17]. The main reason for this is physical side-effects in the CPU, for example, glitches and transitions, which lead to unintended combinations of shares during execution. For example, Coron et al. [Cor+12] show that when a share is overwritten by another share of the same sensitive variable, the power consumption correlates with the combination of both, leading to leakage. In practice, this can be observed when overwriting shares stored in a CPU register or the SRAM. Gigerl et al. [Gig+21; GPM21] report that glitches within the control logic used to address the read/write logic

of the CPU register file might make leakage-free register file accesses impossible. Additionally, they show that shares of the same sensitive variable must not be read or written consecutively, independently whether they are stored in the register file or memory, due to transitions on the respective read/write ports. As a result, they construct a side-channel hardened version of the RISC-V Ibex core (*secured* Ibex), which allows leakage-free execution of masked software in bare-metal mode as long as a few simple software constraints are followed. In our work, we exclusively work with masked software implementation following these constraints and use the *secured* Ibex core as a reference platform for our experiments.

2.3. Coco

In order to evaluate the security of masked cryptographic software, one can either apply empirical or formal verification methods. Empirical verification involves manually taking power measurements of CPUs during computation, followed by statistical analysis that tries to extract sensitive information such as cryptographic keys [CRR02; KJJ99]. The main downside of this approach is the inability to identify the exact source of leakage in a system, i.e., there is no possibility to determine if a leak was caused by the CPU microarchitecture or the masked software implementation. Alternatively, one can use the recently published tool Coco [Gig+21] to formally verify the security of masked software executions in the time-constrained probing model on the gate-level netlist of a CPU. Coco allows to identify concrete gates/wires/registers in the CPU netlist as leakage sources.

In general, Coco takes as input a masked assembly implementation backed up with some annotations and the description of a CPU as a gate-level netlist and then reports whether the execution is leakage-free or not. The annotations (labels) indicate which registers/memory locations contain shares or fresh randomness at the start of the software execution. Internally, the tool then starts by simulating the execution of the software on the CPU hardware in order to obtain an execution trace, which contains a concrete value for each control signal in the CPU. The verifier then propagates the annotated labels through the CPU netlist cycle by cycle while considering the control signals of the execution trace. If Coco finds a gate in a specific cycle that combines all shares of the same sensitive value, the gate in the netlist and the cycle is reported as a leak. Using this information, one can then easily find out whether the leak was caused by the software itself, or micro-architectural side-effects of the CPU. For more details on the internal working mechanisms of Coco, we refer to the original publication [Gig+21].

2.4. Embedded operating systems

Embedded systems requiring multitasking make use of an embedded OS, which runs multiple tasks and manages shared resources such as execution time. Some

scenarios additionally require real-time capabilities, i.e., that the OS guarantees that specific tasks or events can be handled in a specific amount of time. Such operating systems are called real-time operating systems (RTOS), on which we focus in this work. Events occurring during the execution of an RTOS are often called interrupts, such as the periodic timer interrupt, which happens at specific intervals, or non-periodic interrupts caused by IO operations or other external events. To maintain its real-time capabilities, the OS must react and handle the interrupt appropriately. Therefore, it activates the scheduler to select the next task to be run and performs a *context switch* or task switch. In order to do so, information related to the task in the *task control block (TCB)* needs to be saved, which also contains the working state (context), including general purpose and floating point registers. During a context switch, the TCB needs to be saved and restored from memory. The memory area which contains the TCB is called the *TCB memory slot*, which is often part of the tasks's working stack assigned by the OS on startup. RTOSs are currently being used in all kinds of applications, including smart watches, traffic light systems, and home energy monitoring. FreeRTOS [Ama22] is among the most popular RTOSs, and is built into, e.g., Amazon's AWS IoT, Tesla's electric cars, and Bosch's smart home sensors. Other famous RTOSs include the open-source systems Zephyr [Pro22], RIOT [Bac+18], TockOS [Lev+17] KataOS [Dev22], but also many closed-source systems like MQX [Sem22] and PikeOS [GMB22].

3. Attacker model

For our study, we consider a threat model in which an attacker has physical access to a cryptographic device that runs masked software within an embedded OS on a microprocessor. Examples of such devices are electronic wallets, smart cards, or authentication tokens. The attacker's goal is to leak the cryptographic key stored on the device, which is used by cryptographic software that already features sufficient protection against standard differential power analysis (DPA) using masking countermeasures. The attacker does not need to know specific details about the attacked device, such as the concrete source code; it is sufficient to know which cryptographic operation is implemented. To perform the attack, the attacker (1) connects an oscilloscope to the device such that power/EM traces can be recorded, (2) triggers the targeted cryptographic operation by sending an appropriate request via a communication interface, and (3) interrupts the operation by sending some other request with a specific delay. Consequently, all interaction with the device is purely passive , and, e.g., no direct control of the OS runtime is required.

Given a scenario in which an attacker can force interrupts during the execution of masked software at specific points in time, thereby causing potential masking-related correctness problems, what remains is to record a sufficient amount of such computations for a DPA attack to work. The concrete number of power

traces required by the adversary highly depends on the noise level of the attacked system, i.e., a combination of the masked software, the embedded OS, and the microprocessor. Nevertheless, we can look at some previous works that study the impact of unintentional combinations of shares due to overwrites of shares in memory or the register port, which is likely to occur during context switches. For example, as shown by Papagiannopoulos et al. [PV17], about 50 000 traces are sufficient on an ATMega163 8-bit microcontroller to detect memory overwrites, which are the basis of our first proposed attack. The authors also study a form of transitions on register file read ports, the basis of our second proposed attack, which can be exploited by only about 5 000 traces. In practice, these numbers can be higher, e.g., if there is some variation in the timing of the execution of the masked software and the following interrupt. This essentially has a similar effect on the performance of DPA attacks as algorithmic hiding countermeasures and hence generally do not increase the amount of required measurements by more than a quadratic factor.

4. Analyzing Context switches in masked software

In this section, we identify common problems that could arise when running masked cryptographic software as a task on an embedded OS. As a starting point, we use assembly implementations following all constraints from [Gig+21], as well as their *secured* Ibex core for our experiments. The assembly implementations are formally verified for correctness in bare-metal mode using Coco. We then manually insert additional assembly instructions at certain locations to represent a realistic context switch routine. We investigate potential leakage using Coco, which finally reveals two major sources of leakage introduced by context switches. In the following, we provide a more detailed description of our experiment setup and the identified problems.

4.1. Experiment setup

We construct a toy example modeling two tasks to demonstrate the general problem of context switches related to masked software. The first task is executing a masked Keccak S-box (used, e.g., in SHA-3, Shake or Ascon), while the other one is performing unrelated non-cryptographic computations. The toy example is then verified when running on the *secured* Ibex core. In the following, we give more details about the concrete software and hardware setup.

The first task (T_{SBOX}) executes a 1st-order masked Keccak S-box implementation protected by Domain-Oriented Masking (DOM) [GMK16]. In general, the implementation splits up the five 32-bit lanes of the implementation into two shares and uses secure DOM multiplication gadgets to mask non-linear operations. The second task (T_{CNT}) executes a non-cryptographic computation which is

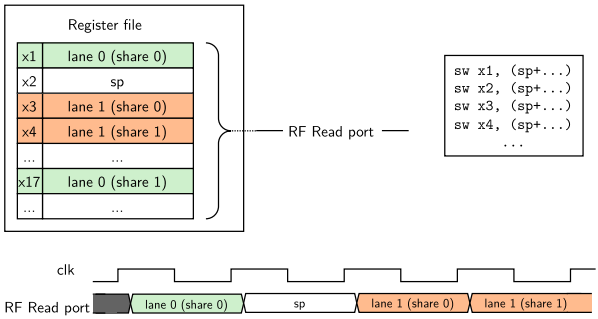


Figure 1.: Transition on register file read port during context switch

unrelated to the first task. We choose a simple function that keeps a counter in a register that is constantly being incremented.

On startup, each task is assigned a specific memory location for the stack and another memory location for the TCB (the TCB memory slot). In practice, the TCB is either stored on the top or at the bottom of the task's working stack. We reserve register r_2 (*sp* on a RISC-V architecture) to store a pointer to this memory area. We then model the effect of real interrupts by manually calling context switching routines from T_{SBOX} and T_{CNT} at certain points in time. This represents a practical scenario where, e.g., an attacker first requests a cryptographic operation via a common communication interface that is then later interrupted by another IO request after a specific amount of time. The context switching routine (context switch) itself is based on an existing implementation included in the FreeRTOS RISC-V port that saves the state of the current task to memory, changes the stack pointer, and then loads the state of the next task from memory. We sketch this function in Appendix A and also further discuss it later in Section 6.1.

We use Coco to investigate the security of our toy example on the *secured* RISC-V Ibex core [Gig+21]. Before starting the experiment, we first verify that the 1st-order Keccak S-box runs securely on the RISC-V Ibex core in bare-metal mode. Consequently, any leakage we observe originates from the context switching activity itself and not from issues with the masked software or the micro-architecture of the Ibex core. In the following sections, we show that the security guarantees derived from verifying bare-metal software no longer hold when executing the masked software within a task.

4.2. Transitions on memory/register file read/write port

Whenever a context switch is performed, the register file contents of the current task are stored to memory register by register in a sequence of store instructions.

Hence, an attacker probing the register file read port or memory write port for the duration of one specific cycle can observe pairs of two register values of the current task. In the second part of the context switch, the register file contents of the next task are loaded from memory, and the current register file contents are overwritten. Again, an attacker probing the register file write port or memory read port can observe pairs of register values of the next task.

If the current task is executing masked software, the register file potentially contains shares of the same secret distributed over several registers. For example, in our toy scenario, the five 32-bit lanes of the Keccak S-box are stored in ten registers, each register containing one share. In Figure 1, we sketch the register file contents of T_{SBOX} at some point during the execution right before a context switch. The timing diagram illustrates the information an attacker can observe by proving the register file read port cycle per cycle. While no critical information can be deduced from the transitions $x_1 \rightarrow x_2$ and $x_2 \rightarrow x_3$, the transition $x_3 \rightarrow x_4$ leaks the Hamming distance between shares 0 and 1 of lane 1, which refers to the unshared value of lane 1.

4.3. Overwriting shares in memory

The exact memory location of the TCB memory slot is defined when the task is created and usually remains unchanged throughout the lifetime of the task on the bottom of the stack. With the general purpose registers being part of the TCB, every context switch during the execution will overwrite the old register contents in memory with the new ones. An attacker probing the respective TCB memory location can therefore observe a transition of the old register value to the new register value. If the memory location previously contained a certain share and is then updated with its counterpart, the attacker can probe the unshared value.

We give an illustration of this scenario using our toy example in Figure 2, in which T_{SBOX} starts execution, is then exchanged by T_{CNT}, until T_{SBOX} resumes. After T_{SBOX}'s second execution, shares stored to memory in the previous context switch might get overwritten by their counterparts. In detail, the following steps occur:

① T_{SBOX} starts execution until it gets interrupted, and the context switch routine is triggered, which saves the register values to the respective TCB memory slot.

② The register file of T_{CNT} is restored, and the task continues execution until the next interrupt.

③ In the context switch, the register values of T_{CNT} are saved to the TCB memory slot of T_{CNT}.

④ The register values of T_{SBOX} are restored from the TCB.

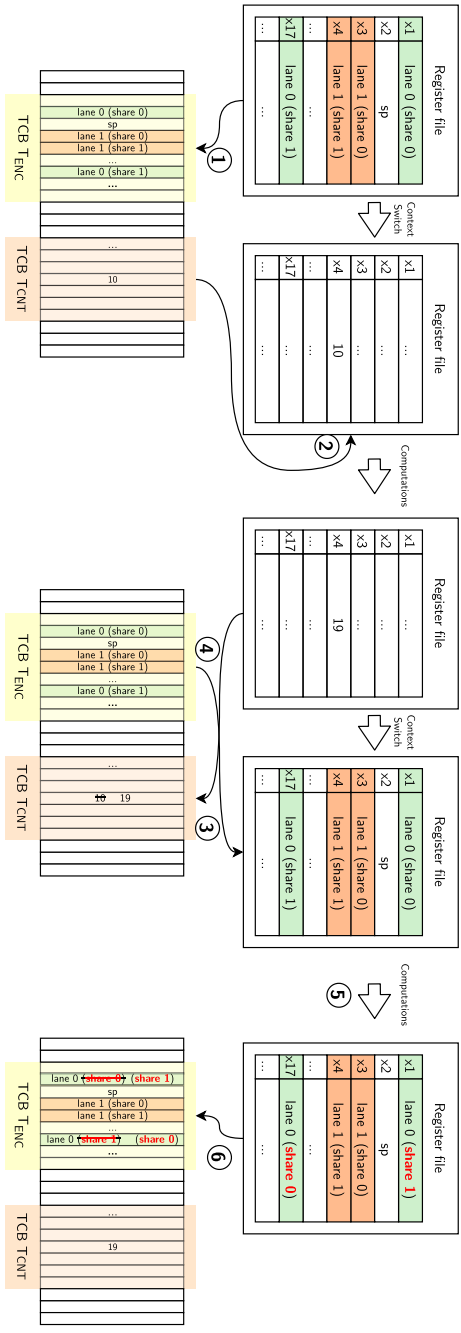


Figure 2.: Overwriting shares in memory during context switch

⑤ T_{SBOX} continues the computation. In our case, the Keccak S-Box implementation exchanges registers x_{17} and x_1 (due to implementation-specific reasons), i.e., the locations of the two shares of lane 0 are swapped in the register file. Note that this can indeed be done securely in an implementation, e.g., with the help of intermediate register clearings.

⑥ During the context switch, the registers are saved, and the old TCB contents of T_{SBOX} are overwritten. More precisely, the memory location storing the old content of register x_1 still refers to share 0 of lane 1. The new content of register x_1 is now - after the computation - the other share, which will, however, be stored in the same memory location. Consequently, share 0 is overwritten with share 1 in memory. The attacker can again observe the Hamming Distance between the old and the new register value, which refers to the unshared lane 1.

4.4. Discussion

Whether the stated problems occur in a practical implementation is still determined by many different parameters, including the frequency of context switches (influenced by the timer interrupt frequency and the attacker's ability to trigger such), the exact point at which the context switch occurs and the exact location of shares in the register file. All these parameters make it infeasible to fix these problems by just adapting the masked software implementation because one would need to take into account the behavior of the embedded OS (such as the sequence in which the registers are spilled) and consider a possible context switch after every instruction. In the next section, we hence aim for more general concepts for secure context switch routines (realized either is software or hardware) that allows masked software, verified in bare-metal scenarios, to preserve security when executed on embedded OSs.

A masked software implementation emits leakage if two shares are combined, e.g., by overwrites and transitions, independently of the concrete masking scheme used. In our case, T_{SBOX} is protected by DOM [GMK16] as an example, but for the analysis, it would be irrelevant which masking scheme is used. In general, DOM has been used both in hardware [GMK16; Gru+21; GSM17; Kni+22; low19], but also in software [Gig+21; GPM21]. Threshold Implementations (TI) [NRR06], Ishai-Sahai-Wagner (ISW) [ISW03], and the Unified Masking Approach (UMA) [GM18] are other examples of masking schemes which have been applied to both hardware and software implementations. Several works on masked software implementations following no specific schemes exist, which are usually optimized for a concrete cryptographic algorithm [Gou+18; GR17; Gro+16; OS05; RP10]. Which scheme to follow depends on the optimization constraints (for example, speed, code size, register sizes, available RNG) of the design. The security of a masked software implementation is, however, not influenced by the concrete technique used because, in any masking scheme, combining two shares will lead

Table 1.: Comparison of basic strategies

Basic strategy	Protection against		Modifications		Overhead	
	Transitions on RW port	TCB memory overwrites	OS	CPU	Memory (TCB)	Runtime (context switch)
No protection	✖	✖	-	-	128 byte	125 cycles
Dummy operations after every load/store	✔	✖	yes	no	128 byte (+0%)	183 cycles (+46%)
Interleaved context switch	✔	✖	yes	no	128 byte (+0%)	125 cycles (+0%)
TCB clearing	✖	✔	yes	no	128 byte (+0%)	183 cycles (+46%)
Rotating TCB memory slots	✖	✔	yes	no	$128\text{ byte} + \frac{128\text{ byte}}{\text{Number of tasks}}$	145 cycles (+16%)
Randomness-refreshed loads and stores (SW)	✖	✔	yes	no	132 byte (+3%)	201 cycles (+61%)
Randomness-refreshed loads and stores (HW)	✖	✔	yes	yes	132 byte (+3%)	143 cycles (+14%)

to leakage.

5. SCA-secure context switching

In this section, we discuss basic strategies to prevent the problems identified in Section 4 and obtain a context switching mechanism that allows masked software, verified in bare-metal scenarios, to preserve security when executed on embedded OSs. The strategies are not specific to a particular embedded OS implementation but should rather give generic concepts which can be integrated into any embedded OS. We provide an in-depth comparison of the different basic strategies, discuss the overhead in terms of memory and runtime, and evaluate their advantages and disadvantages. We formally prove that each strategy allows leakage-free context switching using Coco by integrating each strategy into the toy example introduced in Section 4. The given basic strategies are divided into two categories, either helping against transitions or memory overwrites. Additionally, we discuss why solving these problems on software level (by increasing the masking order) is neither efficient nor feasible in practice.

In Table 1, we give an overview of the strategies, which we will in the following describe more in detail. The table shows for each strategy which problem is addressed, whether modifications are necessary on OS- or CPU-level, and states the overhead compared to the plain, unprotected context switch, which takes 125 cycles to execute on the *secured* Ibex core. To determine the memory overhead, we compare the size of the (potentially modified) TCB to the original TCB (128 byte).

5.1. Basic strategies against transitions

In Section 4, we identified the problem of transitions on memory/register file read/write ports. In the following, we discuss two strategies to prohibit this problem, dummy operations after every load/store and interleaved context switches. In addition to verification of these strategies in the toy example, we add a second verification scenario to strengthen the proposed security guarantees. In the second scenario, we label all 28 registers as shares of the same native value, perform the hardened context switch with these registers and then check if the execution provides 27th-order security. By that, we can show that the constructed secure context switch is indeed SCA-secure independently of the concrete location of shares in the register file.

Dummy operations after every load/store

The most simple solution to prevent transitions between shares on read/write ports is to insert dummy operations, such as `nop` instructions, after every load or store in the context switch. This ensures that the read/write port is always pulled to zero between two memory accesses, preventing direct transitions between shares. While this solution is very simple in terms of integration, its effectiveness and runtime overhead is strongly determined by the underlying CPU microarchitecture. On the *secured* Ibex core, it suffices to put a single `nop` instruction between two memory accesses, yielding a runtime overhead of 46%. However, as shown in works like [GPM21], more complex architectures might require more dummy operations to prevent such leakages.

Instead of using `nop` instructions, one could try to use instructions of the interrupt handling logic which is executed after storing the register contents to the TCB. While this solution would make the context switching more efficient, the feasibility of integrating this is into an embedded OS is highly dependent on the existing context switching/scheduling logic.

Interleaved context switch

A context switch first stores the TCB of the current task selects the next task, and then loads the TCB of the next task. We alter the sequence of these three events to perform an *interleaved* context switch, which first selects the new task, and then loads/stores the contents of the two involved TCB blocks in an alternating (interleaved) manner. The interleaved context switch essentially uses the load operations as dummy operations mentioned in the previous paragraph. On assembly level, this boils down to alternating store and load instructions, as sketched in Appendix B.

While this solution requires no additional runtime or memory overhead, it is very restrictive on the task selection logic, i.e., the scheduler, since all registers used there must not be used during the task's execution. For example, consider a task getting executed, which stores some data into register x_{10}. When it gets

interrupted, the scheduler is triggered to select the next task, and if it uses register x_{10} to do so, the task's data in the register will inevitably be overwritten and therefore lost. Therefore, we consider this solution suitable for our toy example where the task selection works in a very simple round-robin fashion but show in Section 6 that it is infeasible for most embedded OS due to the complexity of the scheduling logic.

5.2. Basic strategies against overwrites

In Section 4, we identified the problem of overwriting shares in memory. In the following, we suggest strategies to prohibit this problem.

TCB clearing

The most naive method to protect against TCB memory overwrites is to clear the TCB of a task executing masked software before saving the registers, i.e., overwriting the memory locations with zeros. For each general-purpose register that is saved during the context switch, this requires one additional store operation. The clearing operations can either be executed in one block before storing the register values or alternating with the actual register store operations, as shown in Appendix C. If the alternating order is used, it also prevents transitions on the register file read port. However, transitions on the register file write port are not prevented. The runtime overhead of a context switch that clears the TCB is 46 %.

Rotating TCB memory slots

On startup, every task is statically assigned a TCB memory slot which is not changed during execution. In order to prevent overwriting shares in memory, one must ensure that the task executing the masked software implementation does not overwrite its own saved registers, which can be done by dynamically changing the TCB memory slot with every context switch. Since physical memory is limited on such constraint devices, allocating a new TCB memory slot with each and every context switch is not feasible. Instead, we need to make sure that TCB memory slots are reused over time. A TCB memory slot can be reused if it does not store the most recent TCB of any currently suspended task (older copies are fine). Consequently, after a task gets interrupted, it must not use its own current TCB memory slot and no memory slot of any other suspended task. We ensure this by adding one additional TCB memory slot such that there is always at least one TCB memory slot that can be reused, and further using a *rotating* assignment of TCB memory slots.

In Figure 3, we sketch this concept using our toy example. In the beginning (①), the two tasks use TCB memory locations TCB #1 and TCB #2 to store their data. We add TCB #3 to ensure one TCB memory location is always

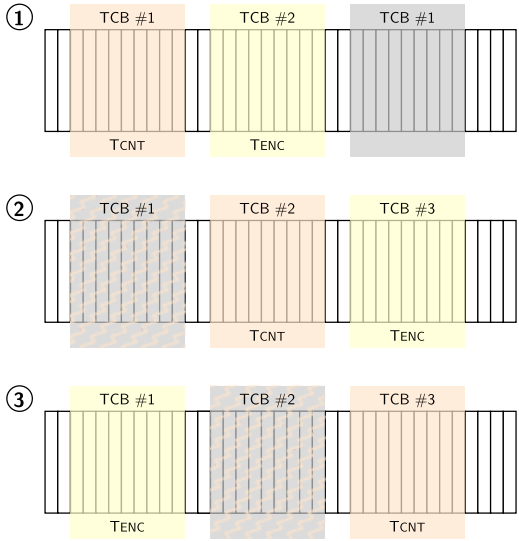


Figure 3.: Rotating TCB memory slots

reusable. After T_{SBOX} has been scheduled once (②), it uses the currently unoccupied TCB #3. It cannot use TCB #1 because it belongs to T_{CNT}, which is currently suspended, and it cannot (re-)use TCB #2 because it contains its own old data, and overwriting might lead to leakage. After T_{CNT} has been executed, it uses TCB #2, which previously belonged to T_{SBOX}, and overwrites the old saved TCB of T_{SBOX}, thus clearing all shares stored to memory. In the next step, T_{CNT} uses TCB #1 to store its TCB after execution, and T_{SBOX} uses TCB #3 (③), leading to a rotating assignment of TCB memory slots.

Although this method comes with almost no time overhead, one additional TCB memory location (128 byte) must be reserved in memory such that the tasks do not overwrite each other's contexts. Additionally, there must be at least one other task running which can overwrite the old saved TCB of the task executing the masked software. If this cannot be ensured, one can either insert a dummy task serving this purpose or extend the kernel in a way such that it clears the old context on purpose, i.e., if no other task was scheduled. It is important to note that the overwrite problem persists even though the task executing the masked software implementation is the only one running in the operating system. Assume that only T_{SBOX} is actively running, and T_{CNT} is sleeping. Whenever T_{SBOX} is interrupted, the working registers will be stored in the TCB. Then, the scheduler will be run and decides that no other task should be scheduled, and loads the register values of T_{SBOX} again. That is, the registers of a task will always be stored to memory when an interrupt occurs, independently of whether

another task will finally be scheduled or not.

Randomness-refreshed loads and stores

TCB memory overwrites can also be counteracted by not storing the plain task context but by adding 32 bits of randomness to each register before storing it to memory. The same randomness can be used for all registers saved in the context switch but must be renewed with every occurring context switch. The randomness must be removed when restoring the context. A task executing masked software will therefore overwrite its old context protected with a different mask. Randomness-refreshed loads and stores can be either implemented in software, by modifying the context switch routine of the OS, or in hardware, by extending the respective CPU core.

We construct a software implementation of this method using our toy example. We assume that there is a certain memory region supplying fresh randomness upon request, which may be connected to an RNG in practice. When a context switch occurs, we first fetch 32 bits of randomness from the location using a load instruction and store it to one of the general-purpose registers. Next, we add the randomness to every register in the register file using a bitwise XOR before saving the register to memory. The used random value is then stored to the TCB of the respective process. When restoring the registers for the process in the next context switch, the previously used random value needs to first be obtained from the TCB again. After issuing the load instruction, which restores a specific GPR value, the random value needs to be added to the GPR register value again in order to obtain the previous value. We sketch this process in Appendix D. The memory overhead of this strategy is around 3% because we need to store the most recently used value of the fresh randomness in the respective TCB, such that it can be fetched before the next load of registers of the respective task. The runtime overhead mostly caused by the additional XOR instructions when storing and restoring the context is 61%. When applying this strategy, at least one register needs to be reserved for storing the randomness, which must not be used in the task's code. Similar to clearing the TCB slot, only tasks executing masked software, such as T_{SBOX}, need to refresh loads and stores in context switches, but all other tasks can stick to the original routine. In practice, the OS usually has a notion about the purpose of each task and can, therefore, easily decide if randomness-refreshing is necessary or not.

Additionally, we provide a hardware implementation of this method, which eliminates most of the runtime overhead by performing the XOR implicitly in the load-store unit of the *secured* Ibex core instead of having to issue a dedicated XOR instruction every time. For this purpose, we extend the core's CSR unit by a 32-bit register which contains the randomness to refresh loads and stores, and a 1-bit register which indicates whether randomness-refreshed loads and stores are enabled. We leave the management of the fresh randomness in memory to the OS, i.e., the OS needs to load fresh randomness from memory to the CSR

register itself. In the context switch routine, one, therefore, needs two additional `CSRW` instructions, one for enabling the countermeasure and one for loading the randomness to the respective CSR register. Therefore, the runtime overhead of such a modified context switch is 14%, as shown in Table 1, of which the most accounts to the management of fresh randomness for both tasks.

5.3. Lazy engineering

In 2015, Balasch et al. [Bal+14] discuss the "lazy engineering" approach of implementing masked software with a protection order that is higher than theoretically required to compensate for a certain loss in practical security due to micro-architectural side-effects. Assuming that a certain masked software implementation is (bare-metal) d-th order secure, a standard context switch routine can generally reduce the security down to $\lfloor d/2 \rfloor$. For example, a transition on a register file read/write port essentially creates leakage that combines values of registers that are loaded/stored in immediate succession. This can, in our case, cut the number of probes required to observe all shares in half, for which one compensates with a higher masking order on level of the masked software.

To achieve a first-order secure masked Keccak S-box, we construct a second-order variant, which provides 1st-order security when using the unprotected context switch. However, the 2nd-order implementation requires much more runtime and randomness: While the 1st-order masked Keccak S-Box needs 174 cycles (without context switches) and 160 bits of fresh randomness, the 2nd-order implementation requires 283 cycles (without context switches), which is an increase of 63%, and 480 bit of fresh randomness (+300%).

Therefore, we do not consider this solution feasible in practice due to the exponential overhead for more complex microarchitectures [GPM21]. Especially for higher masking orders, the overhead caused by lazy engineering grows with the masking order, while the overhead of the other suggested solutions is the same independently of the order.

6. Case Study

In this section, we investigate the security of masked software running as a task in FreeRTOS. In Section 6.1, we introduce our evaluation environment. In Section 6.2, we discuss that the problems identified in Section 4 can indeed be found in practice in FreeRTOS using Coco, and how the context switch in FreeRTOS can eventually be fixed against these problems. In Section 6.3, we provide combinations of the basic strategies which defend against both transitions *and* overwrites and evaluate their performance overhead. While a single hardened context switch comes with some overhead, the overall system overhead is rather negligible, given that the frequency of context switches is usually low in practice. To the best of our knowledge, this is the first analysis of masked software within an

OS, especially on such a level of detail. This is likely due to the considerable effort of creating a suitable analysis environment, which we describe in the beginning of this section. In our case, for example, this involves porting the entire FreeRTOS to the RISC-V Ibex core, adding peripherals to trigger timer interrupts, and simulating the synthesized processor netlist when executing the OS in a suitable simulator for formally verifying it with Coco. We plan to make our evaluation setup, along with all software artifacts, available in a public repository.

FreeRTOS FreeRTOS is a popular open-source embedded OS used in many different IoT projects supported by a large community, which makes it the third most used OS in 2019 [Asp19]. It has been ported to different hardware architectures and platforms, including ARM, RISC-V, and x86-32 [Ser22a], and targets small single-core microprocessors in embedded systems. In order to provide multitasking, the FreeRTOS kernel uses the scheduler to assign processing time to tasks. The scheduler is usually triggered by a (timer) interrupt or the `yield` system call and selects the next task according to a certain policy that takes into account task priorities and deadlines. For our case study, we use the standard preemptive scheduling policy (`configUSE_PREEMPTION = 1` and `configUSE_TIMESLICING = 0`), which means that the task with the highest priority will be selected by the scheduler [Ser22b].

6.1. Evaluation setup

Ultimately, our evaluation environment should allow both formal verification and cycle-accurate performance evaluations. Hence, we need to simulate FreeRTOS, including tasks when running on the *secured* Ibex core in order to obtain a cycle-accurate execution trace. Unfortunately, there exists no exact demo allowing such a simulation, which is why we first need to port FreeRTOS to run on the *secured* Ibex core. We base our port on the existing 32-bit RISC-V Spike simulator demo and make several adaptions, such as changing the addresses of the `mtime` and `mtimecmp` registers. FreeRTOS can only be executed properly in the presence of a periodic timer interrupt. We use a dedicated hardware module for creating these interrupts, which is connected to the *secured* Ibex core. This hardware module provides access to the `mtime` and `mtimecmp` control registers. From a verification perspective, and compared to the bare-metal case, the interrupt signals provide just another set of control signals beside the executed software. Our complete workflow can be sketched as follows:

1. We synthesize the *secured* Ibex core with Yosys [Wol16] to obtain a gate-level netlist in Verilog format.
2. We compile FreeRTOS, including all tasks which are later executed.
3. We wrap the synthesized Ibex netlist into a testbench which includes a timer and a memory model.

4. We simulate the testbench with Verilator [Sny22], which produces a cycle-accurate execution trace. This execution trace contains a concrete value for each control signal in the netlist and can, therefore, directly be used for performance evaluations.

5. In order to do formal verification with Coco, we additionally create the respective annotations (labels) indicating the location of shares/fresh randomness at the beginning of the execution and give these annotations, the netlist of the *secured* Ibex, and the execution trace to Coco.

Besides the execution environment, another central part of the evaluation is the tasks that are executed by FreeRTOS. In order to demonstrate that the identified problems occur, we focus on the same scenario as in the previous sections, i.e., that the OS runs one task executing a masked 1st-order Keccak S-box (T_{SBOX}), and one task which increments a counter (T_{CNT}).

In order to perform a meaningful performance evaluation of our solution, we, however, stick to a larger scenario including a complete Ascon round (T_{ENC}) [Dob+16], which uses the Keccak's S-box core, running beside T_{CNT}. A complete Ascon round is likely to be interrupted more often than a single Keccak S-box by a periodic timer interrupt or external interrupts. Since the performance overhead of our countermeasures stems purely from the context switching, the results become more significant. Similar to our toy example, both T_{SBOX} and T_{ENC} load the input data (shares) from a predefined memory location, compute the S-box/Ascon round, and then stores the input data back to the memory.

6.2. Hardening the FreeRTOS context switch

In this section, we describe the challenges of integrating the basic strategies into FreeRTOS. Protection against both leakage sources in FreeRTOS is achieved by combining the basic strategies against transitions with those against overwrites.

Preventing transitions Besides the problem discussed in Section 4, we could identify a second leakage source caused by transitions in the FreeRTOS scheduler. A context switch can generally be divided into three phases: (1) storing the TCB of the current task, (2) selecting the next task, and (3) loading the TCB of the next task. FreeRTOS uses the same code for phases (1) and (3) as we used in our toy example but has a slightly more complex scheduler, which potentially creates another source of transition-induced leakage between shares on the register file write port. For example, our implementation of the Keccak S-Box was interrupted at a point where registers $x20$ and $x24$ each contained a share of the same native value. The scheduling logic (phase (2)) contains a section of code that first overwrites $x20$, and in the next cycle, overwrites $x24$, causing a leaking transition on the register file write port. Whether these leaks occur is, however, highly dependent on both the concrete scheduler logic and

Figure 4.: Original (left) vs adapted (right) memory layout of FreeRTOS to support rotating TCB memory slots

the masked software implementation. A generic solution can only be achieved when ensuring that the general purpose register values of the task are not used anymore in the scheduling logic, which is why we clear the registers after storing them to memory.

In Section 5, we discuss interleaved context switches as one possible countermeasure against transition leakage, which requires selecting the new task (phase 2) *before* performing storing and loading register values in an interleaved manner. It is not feasible to integrate this strategy into FreeRTOS because the scheduling logic would inevitably overwrite many unsaved task registers when running it before phase (1). Therefore, we instead focus on dummy operations and clearing registers to defend against transition leakage.

Preventing overwrites We include all four basic strategies to prevent overwrites of shares in the TCB. Including rotating TCB memory slot requires the most changes, as the original version of FreeRTOS stores the TCB control data and the values of the general-purpose registers separately. That is, the control data is stored on the bottom of the user stack, and the registers on top. We sketch this in the left part of Figure 4. Hence, the memory location where the registers are stored possibly changes with every context switch, depending on the height of the user stack. This makes it impossible to implement TCB memory slot rotation because when a task uses its stack, e.g., during a function call, it potentially overwrites another task's saved registers which were stored there. Instead, we adapt the layout such that the saved register values are also stored below the working stack of the task and can, therefore, not be overwritten by a task's stack usage (c.f. right of Figure 4). Another challenge is constructing a function find to the next free TCB memory slot before saving any general-purpose registers of the task. The function must not use any general-purpose registers, which still contain the task's data, because overwriting them would inevitably destroy the task's state. Thankfully, we can use registers x_3 and x_4, which are never saved during a context switch and are generally unused in FreeRTOS. The original purpose of these registers is to store the thread pointer and global pointer for optimizations, which is, however, not supported by FreeRTOS.

Fewer system changes are needed to implement randomness-refreshed loads and stores, as we simply extend the TCB struct of the OS by a new variable

Table 2.: Evaluation of variants of SCA-hardened context switching in FreeRTOS

	Runtime in cycles				**Number of context switches**	**Cycles per context switch**
	Total	T_{ENC}	T_{CNT}	**Context switching**		
No protection	4149	1359	1221	1569	4	393
Basic strategies						
Dummy operations + clear registers	5810	1366	1564	2880	6	480 (+22%)
TCB clearing	5729	1364	1648	2717	6	453 (+15%)
Rotating TCB memory slots	4203	1353	1154	1696	4	424 (+7%)
Randomness-refreshed loads and stores (SW)	5836	1395	1587	2854	6	475 (+20%)
Randomness-refreshed loads and stores (HW)	4263	1411	1152	1700	4	425 (+8%)
Combined strategies						
Dummy operations + clear registers + TCB clearing	5978	1376	1395	3207	6	534 (+36%)
Dummy operations + clear registers + rotating TCB memory slots	5880	1349	1478	3053	6	508 (+29%)
Dummy operations + clear registers + Randomness-refreshed loads and stores (SW)	7651	1416	1782	4453	8	557 (+41%)
Dummy operations + clear registers + Randomness-refreshed loads and stores (HW)	5940	1410	1476	3054	6	509 (+29%)

`task_rand` which is updated during the context switch with the used randomness. Also in this case we make use of x_3 and x_4 to load, store and xor the respective random value.

6.3. Discussion

Table 2 shows the overhead of an SCA-hardened context switch when used in FreeRTOS. To measure the performance overhead, we stick to a complete Ascon round (T_{ENC}), scheduled alternating with T_{CNT}. The execution is interrupted by a periodic timer, which frequency can be controlled from software using the `configTICK_RATE_HZ`-define in FreeRTOS. The original configuration of FreeRTOS specifies a timer interrupt every 100 000 cycles which seems plausible considering that in a real system, context switches will not only be triggered by timers but also by many more (non-periodic) external interrupts. As there are no external interrupt sources in our evaluation environment we configure the timer interrupt to occur every 1000 cycles, which however represents an extremely-high-load scenario for the system. Given these numbers, one can then easily extrapolate overheads for scenarios with less frequent context switches.

For each evaluated scenario, we give the total number of cycles needed to compute a full Ascon round, which is the sum of cycles consumed by T_{ENC} and T_{CNT}, and cycles spent on context switching. Note that the number of cycles between two timer interrupts is always constant (1000 cycles), and as the context switching requires more time, less execution time will be available for the tasks, and therefore, more context switches will be necessary in total. We also give the number of cycles required per context switch and the runtime overhead in percent w.r.t. to the basic scenario (no protection).

Which combination to choose depends on the concrete use case. If OS and hardware modifications should be kept minimal and low runtime is not so critical, one should stick to the first option (TCB clearing) because it is very simple to integrate into any OS. If performance is more critical, rotating TCB memory slots are the best option, although they require more OS changes and require at least one other actively running task. Randomness-refreshed loads and stores are more efficient when implemented with hardware support. In software, there is no clear advantage compared to TCB clearing or rotating TCB memory slots. As discussed above, in practice, interrupts are, however, expected to occur much less frequent than in this case study. We argue that therefore, all of the four suggested combinations would be suitable because the amount the total runtime of an Ascon round is increased by applying a secure context switch when interrupted only once is negligible.

Optimizations Further optimization of the runtime of the individual basic strategies is most likely not possible on software level, since they are already written in Assembly language. However, hardware support for clearing the TCB and all registers could be added and would likely result in a performance gain,

although the hardware changes are expected to be much larger than the ones suggested for randomness-refreshed loads and stores. Instead, one could aim for further optimizations on OS level. In fact, the suggested protections must only be applied when a task executing masked software is involved in a context switch. Otherwise, the unprotected (and comparably cheap) context switch can be executed. An additional flag in the task's TCB can be used to distinguish tasks executing masked software from other unrelated tasks.

6.4. Other RTOS

In the following we briefly discuss other open-source RTOS and the possible security implications on masked software running in a task.

Zephyr, RIOT Zephyr [Pro22] is an RTOS maintained by the Linux Foundation for resource-constrained devices with a strong focus on security. RIOT [Bac+18] is similar to Zephyr but comes with different scheduling strategies and supported platforms. The Zephyr and RIOT context switching routines apply a different register order compared to FreeRTOS when storing and loading the register values, which shows why one could never make assumptions about such aspects when designing masked software. We expect both OS are vulnerable to the problems discussed in Section 4, and that our countermeasures can be integrated in a similar way.

TockOS, KataOS TockOS [Lev+17] and the recently announced KataOS [Dev22] are written almost completely in Rust, and both are used for Google's OpenTitan project, which runs the RISC-V Ibex core. The nature of the TockOS context switch suggests the same problems as identified above, which can be solved using the basic strategies except for rotating TCB memory slots. TockOS keeps all processes isolated from each other using a hardware Memory Protection Unit (MPU). Rotating TCB memory slots requires the existence of a common memory region which stores data (register values) of multiple processes, which hurts the principle of isolation, while the other suggested countermeasures are compatible with such isolation techniques.

7. Conclusion

In this paper, we provide the first security analysis of context switches for masked cryptographic software. After showing the fundamental problems created by context switches on embedded OSs, we propose several different mitigation strategies in hardware or software. Ultimately, our hardened context switching routines allow masked software from previous works, verified for security in bare-metal execution, to remain secure when being executed on embedded OSs. We present a case study focusing on FreeRTOS, a popular embedded OS for embedded

devices, running on a RISC-V core, allowing us to evaluate the practicality, ease of integration, and performance of each strategy. While the runtime of hardened context switches is certainly noticeable, we expect the overall impact on system performance to be rather negligible unless the frequency of context switches is very high.

Acknowledgements

This work was supported by the FWF SFB project SpyCoDe F8504, and the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We thank the anonymous reviewers for their valuable suggestions and comments.

Appendix A. Unprotected context switch

```
task_switch:
  sw x1, (sp)
  sw x2, 4(sp)
  sw x3, 8(sp)
  sw x4, 12(sp)
  # ...
  # Select next task
  # ...
  lw x1, (sp)
  lw x2, 4(sp)
  lw x3, 8(sp)
  ...
  ret
```

Appendix B. Interleaved context switch

```
task_switch_interleaved:
  mv sp, x30    # Reserve x30, never use in code
  # ...
  # Select next task
  # ...
  sw x1, (x30)
  lw x1, (sp)
  sw x2, 4(x30)
  lw x2, 4(sp)
  sw x3, 8(x30)
  lw x3, 8(sp)
  sw x4, 12(x30)
  lw x4, 12(sp)
  ...
  ret

```

Appendix C. TCB clearing

```
task_switch_clear_tcb:
  sw x0, (sp)      #x0 is constantly tied to 0
  sw x1, (sp)
  sw x0, 4(sp)
  sw x2, 4(sp)
  sw x0, 8(sp)
  sw x3, 8(sp)
  sw x0, 12(sp)
  sw x4, 12(sp)
  # ...
  # Select next task
  # ...
  lw x1, (sp)
  lw x2, 4(sp)
  lw x3, 8(sp)
  ...
  ret

```

Appendix D. Randomness-refreshed loads and stores (SW)

```
task_switch_rand_refresh_sw:
  li x30, addr_prng
  # Reserve x30, never use in code
  lw x30, (x30)
  # x30 now contains fresh randomness
  xor x1, x1, x30
  sw x1, (sp)
  xor x2, x2, x30
  sw x2, 4(sp)
  xor x3, x3, x30
  sw x3, 8(sp)
  xor x4, x4, x30
  sw x4, 12(sp)
  # Store x30 to TCB
  # ...
  # Select next task
  # ...
  # Load randomness used in previous store from TCB to x30
  lw x1, (sp)
  xor x1, x1, x30
  lw x2, 4(sp)
  xor x2, x2, x30
  lw x3, 8(sp)
  xor x3, x3, x30
  ...
  ret

```

References

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. “An Implementation of DES and AES, Secure against Some Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International*

Workshop, Paris, France, May 14-16, 2001, Proceedings. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 309–318.

[AG21] Farhad Andalibi and Paulo Garcia. "Near-Native Interrupt Latency in Real-Time Guests: Handler Emulation Through Memory Map Morphing". In: *ICCDE 2021: 7th International Conference on Computing and Data Engineering, Phuket, Thailand, January 15 - 17, 2021*. ACM, 2021, pp. 94–98.

[Ama22] Inc. Amazon Web Services. *FreeRTOS*. https://www.freertos.org/. Retrieved on December 15th, 2022. 2022. URL: https://www.freertos.org/.

[Asp19] Aspencore. *2019 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments*. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, Retrieved on November 3rd, 2022. 2019.

[Bac+18] Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet Things J.* 5.6 (2018), pp. 4428–4440.

[Bal+12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. "Theory and Practice of a Leakage Resilient Masking Scheme". In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 758–775.

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*. Vol. 8968. Lecture Notes in Computer Science. Springer, 2014, pp. 64–81.

[Bal+15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. "DPA, Bitslicing and Masking at 1 GHz". In: *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 599–619.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bar+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 189–228.

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. "Analysis and Comparison of Table-based Arithmetic to Boolean Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 275–297.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 397–426.

[Bel+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In: *EUROCRYPT (3)*. Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–341.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. "Improving interrupt response time in a verifiable protected microkernel". In: *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. Ed. by Pascal Felber, Frank Bellosa, and Herbert Bos. ACM, 2012, pp. 323–336.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 82–98. DOI: 10.1007/978-

3-319-89641-0_5. URL: https://doi.org/10.1007/978-3-319-89641-0%5C_5.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 69–81.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[Dev22] AmbiML Developers. *KataOS*. https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html. 2022.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2. Submission to the CEASAR Competition*. https://ascon.iaik.tugraz.at/files/asconv12.pd. Retrieved on February 4th, 2021. 2016.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei,*

Taiwan, September 25-28, 2017, Proceedings. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GM18] Hannes Groß and Stefan Mangard. "A unified masking approach". In: *J. Cryptogr. Eng.* 8.2 (2018), pp. 109–124.

[GMB22] SYSGO GMBH. *PikeOS*. https://www.sysgo.com/pikeos. Retrieved on December 14th, 2022. 2022. URL: https://www.sysgo.com/pikeos.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Gou+18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. "Secure Multiplication for Bitslice Higher-Order Masking: Optimisation and Comparison". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 3–22.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 158–172.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors". In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GR17] Dahmun Goudarzi and Matthieu Rivain. "How Fast Can Higher-Order Masking Be in Software?" In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Lecture Notes in Computer Science. 2017.

[Gro+16] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. "Bitsliced Masking and ARM: Friends or Foes?" In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*. Vol. 10098. Lecture Notes in Computer Science. Springer, 2016, pp. 91–109.

[Gru+21] Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. "DOMREP-An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 4321–4335.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 2017, pp. 205–212.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[Kni+22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. "Automated Generation of Masked Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 589–629.

[Lev+17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP'17. Shanghai, China: ACM, Oct. 2017, pp. 234–251. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132786. URL: http://doi.acm.org/10.1145/3132747.3132786.

[low19] lowRISC contributors. *AES HWIP Technical Specification*. https://opentitan.org/book/hw/ip/aes/index.html. Retrieved on 19/4/2023. 2019. URL: https://opentitan.org/book/hw/ip/aes/index.html.

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1297.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 199–216.

[MPW22] Ben Marshall, Dan Page, and James Webb. "MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 175–220. DOI: 10.46586/tches.v2022.i1.175-220. URL: https://doi.org/10.46586/tches.v2022.i1.175-220.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[OS05] Elisabeth Oswald and Kai Schramm. "An Efficient Masking Scheme for AES Software Implementations". In: *Information Security Applications, 6th International Workshop, WISA 2005, Jeju Island, Korea, August 22-24, 2005, Revised Selected Papers*. Ed. by JooSeok Song, Taekyoung Kwon, and Moti Yung. Vol. 3786. Lecture Notes in Computer Science. Springer, 2005, pp. 292–305.

[Pro22] Zephyr Project. *Zephyr OS*. https://www.zephyrproject.org/. Retrieved on December 14th, 2022. 2022. URL: https://www.zephyrproject.org/.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. "Mind the Gap: Towards Secure 1st-Order Masking in Software". In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 282–297.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427.

[Sem22] NXP Semiconductors. *MQX Real-Time Operating System (RTOS)*. https://www.nxp.com/design/software/embedded-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS. Retrieved on December 14th, 2022. 2022. URL: https://www.nxp.com/design/software/embedded-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS.

[Ser22a] Amazon Web Services. *FreeRTOS Kernel Ports*. https://www.freertos.org/RTOS_ports.html. Retrieved on December 5th, 2022. 2022. URL: https://www.freertos.org/RTOS_ports.html.

[Ser22b] Amazon Web Services. *FreeRTOS Scheduling*. https://www.freertos.org/implementation/a00005.html. Retrieved on December 5th, 2022. 2022. URL: https://www.freertos.org/implementation/a00005.html.

[She+21a] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. "Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 685–699.

[She+21b] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[Sny22] Wilson Snyder. *Verilator*. https://www.veripool.org/wiki/verilator. Retrieved on February 2nd, 2021. 2022.

[Wan+15] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. "Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON". In: *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*. Ed. by Kaisa Nyberg. Vol. 9048. Lecture Notes in Computer Science. Springer, 2015, pp. 181–198.

[Wol16] Claire Wolf. *Yosys Open Synthesis Suite*. http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021. 2016.

[ZLG09] Peifeng Zhang, Hong Li, and Zhigang Gao. "PIL: A Method to Improve Interrupt Latency in Real-Time Kernels". In: *International Conference on Scalable Computing and Communications / Eighth International Conference on Embedded Computing, ScalCom-EmbeddedCom 2009, Dalian, China, September 25-27, 2009*. Ed. by Keqiu Li, Geyong Min, Yongxin Zhu, Meikang Qiu, and Wenyu Qu. 2009.

6

Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency

Publication Data. Barbara Gigerl, Franz Klug, Stefan Mangard, Florian Mendel, and Robert Primas. "Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency".
In: *TCHES*. 2024.

Contribution. The author of this thesis did the hardware implementation and all the area estimates as well as the empirical and formal security evaluations, and wrote the text for the paper. Furthermore, the author of this thesis contributed to the construction of the implemented AES S-box design and the methods for reducing the randomness requirements.

Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency

Barbara Gigerl[1], Franz Klug[2], Stefan Mangard[1], Florian Mendel[2], Robert Primas[3]

[1] Graz University of Technology [2] Infineon Technologies AG [3] Intel Labs

Abstract Cryptographic devices in hostile environments can be vulnerable to physical attacks such as power analysis. Masking is a popular countermeasure against such attacks, which works by splitting every sensitive variable into $d + 1$ randomized shares. The implementation cost of the masking countermeasure in hardware increases significantly with the masking order d, and protecting designs often results in a large overhead. One of the main drivers of the cost is the required amount of fresh randomness for masking the non-linear parts of a cipher. In the case of AES, first-order designs have been built without the need for any fresh randomness, but state-of-the-art higher-order designs still require a significant number of random bits per encryption. Attempts to reduce the randomness however often result in a considerable latency overhead, which is not favorable in practice. This raises the need for AES designs offering a decent performance tradeoff, which are efficient both in terms of required randomness and latency.

In this work, we present a second-order AES design with the minimal number of three shares, requiring only 3 200 random bits per encryption at a latency of 5 cycles per round. Our design represents a significant improvement compared to state-of-the-art designs that require more randomness and/or have a higher latency. The core of the design is an optimized 5-cycle AES S-box which needs 78 bits of fresh randomness. We use this S-box to construct a round-based AES design, for which we present a concept for sharing randomness across the S-boxes based on the *changing of the guards* (COTG) technique. We assess the security of our design in the probing model using a formal verification tool. Furthermore, we evaluate the practical side-channel resistance on an FPGA.

1. Introduction

Embedded devices running cryptographic hardware implementations need to be protected against physical attacks, such as differential power analysis [KJJ99], in which an attacker observes the power consumption of the device and uses the information to learn about secret values, e.g., the cryptographic key. Masking

is a popular approach to protect against these attacks on implementation level, aiming at making the power consumption independent of the processed sensitive value [Cha+99]. To protect against a d-th order DPA attack, masking splits each sensitive value into $d + 1$ shares such that an attacker probing up to d shares cannot recover the sensitive value.

Applying the masking countermeasure to a cryptographic hardware implementation comes with a considerable area overhead, which increases significantly with the masking order d [GIB18; ISW03; MRB18; Nag+22; SP06]. This overhead is not only caused by an increased area for the handling of the shares but also by the increased demand for fresh randomness that needs to be generated and distributed for masking the non-linear parts of the cipher. While the linear parts can be computed by evaluating them for each share individually, the non-linear parts, such as S-boxes, need to operate on several shares at once and, therefore, require randomness for refreshing to prevent unmasking of intermediate computation results, especially in the presence of glitches [Bar+17; Bel+17; GMK16; ISW03; Rep+15]. The need for fresh randomness usually goes hand-in-hand with an increased design area caused by the required random number generator (RNG) instances.

Methods to reduce randomness for a masked design have been studied extensively, especially focusing on AES. Since its selection by NIST in 2000, the AES [Nat01] has become an essential component for many cryptographic applications in industry. While the first proposed first-order sharings of the AES required about 3 000 to 5 000 random bits per encryption, there by now exist several works suggesting how to perform the computation without any fresh randomness [SM21; Sug19; WM18]. Compared to that, higher-order masked AES designs still require a significant amount of fresh randomness and area. While first works on second-order masking of the AES in hardware require more than three shares [Bey+21; Cnu+15], in 2016, De Cnudde et al. [Cnu+16] propose an S-box design with three shares which needs 162 fresh random bits and has a latency of five cycles, resulting in 19 440 random bits and 276 cycles per encryption. Gross et al. [GMK16] improve this situation by proposing a 5-cycle S-box protected by DOM (Domain-Oriented Masking) with only 84 random bits, resulting in 16 800 random bits and 200 cycles per encryption. Reducing the amount of randomness for a design however comes at the price of latency. Naturally, less randomness implies fewer capabilities to control the effect of glitches in a circuit, which in turn needs to be compensated for with more register stages, leading to a higher latency. For example, Dhooghe et al. [DSM22] recently show how to construct a second-order masking of the AES with only 1 012 fresh random bits per encryption, which however result in an S-box latency of 9 cycles per round. In recent years, low-latency has been generally identified as an important design goal for masked designs. Several works construct masked designs optimized for extremely low cycle counts [GIB18; Nag+22; Sas+20; Sim+22]. For example, Gross et al. [GIB18] propose a second-order masked low-latency DOM-AES

S-box, which only needs two cycles per round but requires almost 900 000 random bits per encryption.

On architectural level, the performance of AES designs can be improved by employing a *parallel* or *round-based* design, in which the S-box is instantiated once per key/state byte, and all instances operate in parallel. By contrast, *serial* designs instantiate the S-box once, which is fed with a new key/state byte in every clock cycle. In parallel designs, the number of pipeline stages in the S-box determines the latency of an encryption round, and therefore, an S-box with a low latency is preferable. While most works in literature focus on serial designs, parallel designs have only been marginally addressed despite their clear practical relevance. For example, Google's OpenTitan project [low19], which aims at building an open root of trust (ROT) chip, includes a parallel AES architecture protected by DOM. They use a first-order version of the 5-cycle DOM AES S-box, which leads to an encryption latency of about 50 cycles per 128-bit block. One of the main challenges when constructing such designs is the high amount of randomness required per cycle, and in practice, it is not trivial to come up with RNGs allowing for such high demands of bandwidth yet keeping the required amount of randomness somewhat balanced per cycle.

Given that first-order protection often does not provide the required security level in practice, and serial designs are often not suitable for the desired performance, the goal is to build second-order designs targeting both low-randomness and low-latency.

Contributions In practice, second-order masked AES designs should be efficient and provide a suitable tradeoff between area and latency, which clearly presumes a three-share design. However, state-of-the-art three-share designs are either optimized for low-latency or for low-randomness. Additionally, given the need for parallel designs, the demands of fresh randomness per cycle of these designs are unevenly distributed and often simply too high. We improve the situation by providing the following contributions:

- We present a second-order masked AES S-box based on DOM, which works with the minimum number of three shares, has a latency of only five cycles, and requires 78 bits of fresh randomness. In order to construct this S-box, we take the original DOM design as a starting point and demonstrate that fixing the flaw in higher-order DOM-*dep* multipliers, as identified by [Moo+19], is possible using more randomness. However, we also show that all DOM-*dep* multipliers can be replaced by more area-efficient adapted DOM-*indep* multipliers, which allows to perform one S-Box computation with 78 bits of fresh randomness. *(Section 3)*
- We propose an efficient parallel AES architecture similar to the one used in OpenTitan with an encryption latency of 51 cycles. We show how one encryption can be computed with only 3 200 bits by applying a special COTG-based concept for reusing randomness across all S-box instances

for the key and plaintext. The 3 200 bits can smoothly be delivered by an RNG with a bandwidth of 64 fresh random bits per cycle. Given the 5-cycle latency per round, our design currently requires the least amount of fresh randomness in literature. *(Section 4)*

- We evaluate our AES design in terms of area and randomness and compare it to other state-of-the-art designs. *(Section 5)*
- Using a formal verification tool, we show the second-order security of our S-box design and investigate the security of our COTG-based sharing concept for key and plaintext for one round. We deploy our design on an FPGA and show that no leakage can be detected with up to 100 million traces. *(Section 6)*
- We provide access to the complete HDL code on GitHub[1].

2. Preliminaries

2.1. Notation

We denote the sharing of a sensitive variable X with $X = (X_0, X_1, X_2)$, i.e., the subscript index denotes a specific share. Every state byte in the AES is described as $s^{(i,j)}$, where i refers to the row index and j refers to the column index, according to the convention introduced in the AES specification [Nat01]. For example, $s_0^{(0,2)}$ refers to the first share (share domain 0) of the state byte in row 0, column 2. Every key byte in the AES is described as $k^{(i,j)}$ accordingly with the sharing $k^{(i,j)} = (k_0^{(i,j)}, k_1^{(i,j)}, k_2^{(i,j)})$.

2.2. Masking

Masking [Cha+99; GP99; ISW03] aims at defeating side-channel attacks that work by randomizing sensitive values by splitting them into $d + 1$ uniformly random shares. An adversary observing (probing) up to d shares cannot deduce any information about the sensitive value. In classical Boolean masking, the sharing of a sensitive variable s given by $(s_0, s_1, ...s_d)$ must satisfy $s = s_0 \oplus s_1 ... \oplus s_d$. The shares $s_0, s_1, ...s_{d-1}$ are randomly sampled from a uniform distribution, while $s_d = s \oplus s_0 \oplus s_1 ... \oplus s_{d-1}$. For example, in a second-order masking scheme ($d = 2$), s is represented by the sharing (s_0, s_1, s_2) such that $s = s_0 \oplus s_1 \oplus s_2$. s_0 and s_1 are chosen uniformly at random and $s_2 = s_0 \oplus s_1$.

Implementing the masking countermeasure for non-linear functions such as the AES S-box, which computes the inversion in $GF(2^8)$, is especially challenging because they require combining all shares of a sensitive value in a secure and correct way. Hardware-related side-effects such as glitches and transitions need to be considered, which could reveal secret information in an otherwise secure masked

[1] https://github.com/barbara-gigerl/aes-secondorder-guards

implementation [ISW03; MPG05; MPO05]. Masking schemes for the AES S-box have been addressed frequently in literature [Bey+21; Cnu+16; DSM22; GMK16; Mor+11; Osw+05; RP10; SP06]. Canright [Can05] presents a decomposition into $GF(2^4)$ and $GF(2^2)$ field elements to perform the inversion more efficiently, which has since then been the basis for many works on masking the AES, including DOM by Gross et al. [GMK16].

2.3. Security Verification of Masking

Empirical measurements are generally an important indicator for the practical security of a masked implementation. However, collecting power traces is usually cumbersome and error-prone, and the results heavily depend on the platform and measurement setup. Formal verification tools represent a complementary approach that allows the analysis of a masked implementation within a specific attacker model, such as the classic probing model [ISW03].

Rebecca [Blo+18] is a formal verification tool to prove the security of masked hardware implementations at any order. It examines the leakage of a given circuit by investigating each gate and determining whether the gate output correlates directly with the unshared sensitive value. Rebecca approximates this correlation using Fourier expansions of Boolean functions [ODo14] and checks for leaks using a SAT solver, making it feasible to verify larger constructions at the cost of accuracy. However, it has been shown that the rate of false positives (tool falsely reports leak) is very low, and false negatives (tool falsely reports no leak) are not possible at all [GPM23]. Other tools like SILVER [KSM20] determine this correlation by exhaustively computing the probability distribution of each gate, which allows a very accurate analysis, but it hardly applies to more complex circuits such as higher-order AES S-boxes [DSM22]. In this work, we will use Coco [Gig+21], a tool based on Rebecca. Coco applies the time-constrained probing model, allowing an adversary to place d probes on an arbitrary wire in the circuit. Each probe allows observing the value of the wire for one specific clock cycle, including transitions and glitches. A masked hardware implementation is considered dth-order secure if the adversary cannot learn any information about the sensitive value by combining the values of these probes.

2.4. Changing of the Guards (COTG)

Masked designs based on TI (Threshold Implementation) require non-completeness and uniformity to be first-order secure [NRR06], but obtaining a uniform output sharing of a masked S-box often requires explicit remasking with fresh randomness. The changing of the guards (COTG) concept was introduced by Daemen [Dae17] to achieve uniformity more efficiently by replacing this fresh randomness with unrelated parts of the cipher state. For example, considering a TI S-box function S and the respective component functions S_0, S_1, S_2 arranged in an S-box layer that maps the shared inputs a, b, c to the shared outputs A, B, C as follows (for

$0 \leq i \leq 2$):

$$A_i = S_0(b_i, c_i) \qquad B_i = S_1(a_i, c_i) \qquad C_i = S_2(a_i, b_i)$$

If the sharings of A, B, C are not uniform one needs to perform resharing, which can either be done with fresh randomness or, as suggested by COTG, with another unrelated input share such as the one of the neighbor S-box (for $0 \leq i \leq 2$):

$$A_i = S_0(b_i, c_i) \oplus b_{i-1} \oplus c_{i-1} \quad B_i = S_1(a_i, c_i) \oplus c_{i-1} \quad C_i = S_2(a_i, b_i) \oplus b_{i-1}$$

The values of b_{-1} and c_{-1} need to be instantiated with fresh random values. COTG has been applied to several TI implementations including AES [Ask+22; Bey+21; DSM22; SBM21; Sug19; WM18], KETJE [ANR19], Ascon and Keyak [SD17], ARX ciphers [JPS18], and PRINCE [MMM21]. The original idea of COTG is to use the input bytes of the right neighbor S-box as guards and use fresh randomness for the last S-box that does not have a right neighbor. In our work, we propose a more complex selection of guards by precisely analyzing which other state bytes are unrelated and which are not, eliminating the explicit need for fresh randomness for the last S-box.

3. Efficiently Masking the AES S-box

In this section, we present a 5-stage pipelined AES S-box with three shares requiring only 78 bits of fresh randomness, which is currently the lowest amount of randomness required for 5-cycle latency. The second-order S-box design DOM [GMK16], which serves as the basis for our design, requires 104 random bits, while the 5-cycle TI-design of De Cnudde et al. [Cnu+16] needs 162 random bits.

In Section 3.1, we describe DOM and the basic structure of their proposed S-box, which uses the Canright decomposition and performs the multiplications in $GF(2^2)$ and $GF(2^4)$ with DOM-*dep* and DOM-*indep* multipliers. In 2019, [Moo+19] pointed out a flaw in higher-order DOM-*dep* multipliers, which we revisit Section 3.2, and discuss a possible fix for this. Unfortunately, including this fix into the second-order S-box requires an additional 20 bits of fresh randomness, resulting in 104 bits in total. Therefore, in Section 3.3, we show how one can optimize the S-box design such that the DOM-*dep* multipliers are not needed anymore at all and can be replaced by three types of adapted versions of DOM-*indep* multipliers, resulting in a randomness-optimized S-box design. We check the second-order security of our S-box design with Coco and give details on the verification in Section 6.

3.1. DOM-based Masking of the AES S-box

In 2016, Gross et al. [GMK16] introduce DOM as a low-cost method to protect circuits against SCA at arbitrary protection orders. DOM is based on the idea of separating shares into independent domains and adding fresh randomness whenever terms from different domains are combined. They introduce a five-cycle variant of the AES S-box intended for high-speed encryption, which serves as the basis of our work and is also used in the OpenTitan project. The S-box design follows Canright's propositions [Can05].

For both the subfield multiplications, Gross et al. propose two masked multiplication gadgets. The second-order DOM-*indep* multiplier, which we will refer to DOM-*indep* multiplier (Type A), is used to multiply two independently shared field elements A with sharing (A_0, A_1, A_2), and B with sharing (B_0, B_1, B_2) using the random variables z_0, z_1, z_2. The resulting output sharing (C_0, C_1, C_2), with registers indicated by parenthesis, is:

$$C_0 = (A_0 \times B_0) \oplus (A_0 \times B_1 \oplus z_0) \oplus (A_0 \times B_2 \oplus z_1) \quad (1)$$

$$C_1 = (A_1 \times B_0 \oplus z_0) \oplus (A_1 \times B_1) \oplus (A_1 \times B_2 \oplus z_2) \quad (2)$$

$$C_2 = (A_2 \times B_0 \oplus z_1) \oplus (A_2 \times B_1 \oplus z_2) \oplus (A_2 \times B_2) \quad (3)$$

The multiplication works in three phases. First, in the *calculation* phase, shares of different domains (cross-domain multiplication) and shares of the same domain (inner-domain multiplication) are multiplied in the respective field. Cross-domain multiplication terms are then refreshed with three fresh random values in the *resharing* phase and stored into a register, while inner-domain terms do not need to be refreshed. In the *integration* phase, the multiplication terms of each component function are accumulated.

In case the multiplier inputs are not shared independently, e.g., when multiplying $A \times A$, one could simply use a DOM-*indep* multiplier and reshare one of its inputs, which however comes at the cost of additional randomness and a register stage. Therefore, Gross et al. propose the DOM-*dep* multiplier that uses a random blinding variable p with the sharing (p_0, p_1, p_2) to compute $A \times B = A \times (B + p) + (A \times p)$. A DOM-*indep* multiplier is used to compute $(A \times p)$, and therefore, the complete second-order DOM-*dep* multiplier requires six fresh random values.

Given these two multiplication gadgets, the 5-cycle S-box first converts the 8-bit input shares from the polynomial basis to the normal basis, inverts them in $GF(2^8)$ by decomposition into $GF(2^4)$ and $GF(2^2)$ field elements, and converts them back. More precisely, in Stage 1, the 8-bit input shares are converted using a linear mapping, which linearly combines the bits of a share within one domain each. Due to glitches, the output of the linear mapping might temporarily result in a related sharing, and therefore, a $GF(2^4)$ DOM-*dep* multiplier is used. In Stage 2, the resulting $GF(2^4)$ field elements are combined with the outputs of the square scalers, and glitches could temporarily produce a related input sharing, therefore requiring the use of a $GF(2^2)$ DOM-*dep* multiplier. In Stage

3, a similar situation occurs, and consequently, both $GF(2^2)$ multipliers must be DOM-*dep* multipliers. The last multipliers in Stage 4 take as an input the pipelined S-box inputs and the output of Stage 3, which are clearly independent of each other, and therefore, $GF(2^4)$ DOM-*indep* multipliers can be used. In Stage 5, the output shares are converted back to the polynomial basis using the inverse linear mapping.

3.2. Fixing the second-order DOM-dep multiplier

In a follow-up work, Moos et al. [Moo+19] point out a flaw in the DOM-*dep* multiplier for $d \geq 2$. Recall from the previous section that a DOM-*dep* multiplier computes $A \times B = A \times (B + p) + (A \times p)$. They show that DOM-*dep* multipliers are not secure in the presence of glitches by combining information about the individual shares of $A \times (B + p)$, and multiplication terms in the DOM-*indep* multiplier (Type A) computing $(A \times p)$. A second-order adversary possesses two probes. One probe is used to access the individual shares of $A \times (B + p)$, which includes $A_2 \times (B_0 \oplus p_0)$. The other probe is placed in the DOM-*indep* multiplier to access the shared subproducts of $(A \times p)$, which includes the cross-domain term $A_1 \times p_0$. By combining these two probed values and considering that the sharings of A and B are related, the adversary can derive information about the sensitive value A.

We propose a way to fix this issue by preventing the adversary from accessing $B_0 \oplus p_0$ directly by adding more randomness to it. More concretely, we refresh the term $B+p$ with a sharing of the zero-bit vector $(q_0, q_1, q_0 \oplus q_1)$ and store that value to a register. The computation performed is now $A \times B = A \times (B+p+0) + (A \times p)$ with 0 being a shared into q_0 and q_1 such that $0 = q_0 \oplus q_1$. Hence, the first probe will only allow access to $A_2 \times (B_0 \oplus p_0 \oplus q_0)$, and no information about A can be inferred due to the random value q_0. The advantage of this solution compared to refreshing B and using a DOM-*indep* multiplier afterward is that no additional register stage is required. Nevertheless, for the fixed second-order DOM-*dep* $GF(2^2)$ multiplier, 16 instead of 12 random bits are needed, or 32 instead of 24 in the case of $GF(2^4)$.

We successfully verify with Coco that our proposed solution indeed solves the issue and is second-order probing-secure in the presence of glitches. Furthermore, we apply the formal verification tool SILVER [KSM20] to prove that our construction is secure under the 2nd-order PINI (Probe Isolating Non-Interference) [CS20] notion and can, therefore, trivially be composed.

3.3. Optimized second-order DOM S-box

Integrating the proposed fix directly into the S-box design requires 104 bits of fresh randomness instead of the originally proposed 84 bits. For a complete AES encryption, this results in 20 800 required random bits instead of 16 800. Therefore, we propose a way to optimize this construction by replacing all

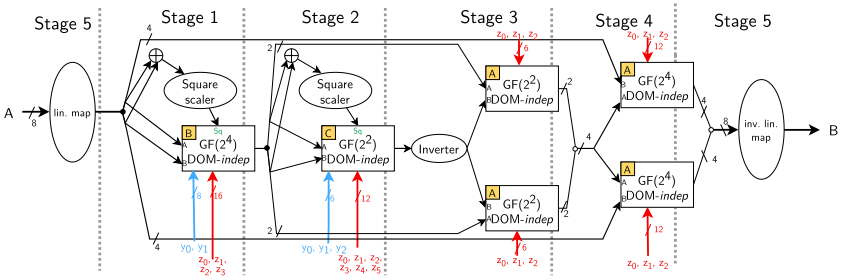


Figure 1.: Our second-order AES DOM S-box with three shares and five register stages, requiring 78 bits of randomness. For simplicity, we draw a single line for all three shares. The • indicates that a signal is split into a lower and upper part. The ∘ indicates that the lower and upper parts of a signal are concatenated. Register stages are sketched by gray dotted lines. The respective type of each DOM-*indep* multiplier is indicated by a letter in the yellow box in the upper left corner, that is either Type A (Equations (1-3)), Type B (Equations (4-6)) or Type C (Equations (7-9)).

DOM-*dep* multipliers with three types (Type A, B, C) of adapted DOM-*indep* multipliers, which are more efficient in both area and randomness. The resulting 78 bits of required fresh randomness are even less than in the originally proposed design. While the Type A multiplier refers to the original DOM-*indep* multiplier, the Type B and C multipliers work by additionally refreshing inner-domain multiplication terms besides cross-domain multiplication terms, which leads to an independent output sharing of a multiplier, and therefore allows the use of a DOM-*indep* multiplier in the next pipeline stage. Figure 1 gives an overview of the complete S-box design. Using Coco, we successfully verify the second-order security of our S-box. Now we describe the design considerations made in each stage in detail.

Linear mapping of input. Our goal is to replace the DOM-*dep* multiplier in Stage 1 with a DOM-*indep* multiplier. DOM-*indep* multipliers require that their inputs (the outputs of the linear map in our case) are shared independently. In general, glitches may temporarily cause a related sharing at the output of the linear map, and therefore, we need to store the output of the linear map in a register. Since our goal is a considerably low latency, we do not add an additional pipeline stage but move the computation of the linear map to the pipeline stage before. Considering the entire AES design, the complete linear layer (including the inverse linear map, ShiftRows, MixColumns, and AddRoundKey) is already computed in Stage 5 (c.f. Section 4), where we now also move the linear map

of the SubBytes computation of the next round. Hence, the state registers in the design will not store the field elements in the polynomial base but the field elements in the normal base. From a security perspective, it is valid to do so because in Stage 5, only linear functions are computed, and adding the linear map to the end will not cause any additional leakage.

Multiplier in Stage 1 (Type B). We want to replace the DOM-*dep* multiplier in Stage 2 with a DOM-*indep* multiplier. The DOM-*indep* multiplier in Stage 2 only supports independent inputs, so the DOM-*indep* multiplier in Stage 1 needs to be modified such that it generates an independent output sharing. To do so, we need to perform the addition of the square scaler already in Stage 1, protect the inner-domain multiplication terms and use additional randomness on the cross-domain multiplication terms. The modified DOM-*indep* multiplier, which will be referred to as the Type B multiplier, used in Stage 1 with parenthesis again indicating registers, is given by:

$$C_0 = (A_0 \times B_0 \oplus Sq_0 \oplus y_0 \oplus y_1) \oplus (A_0 \times B_1 \oplus z_0 \oplus z_3) \oplus (A_0 \times B_2 \oplus z_1) \quad (4)$$
$$C_1 = (A_1 \times B_0 \oplus z_0) \oplus (A_1 \times B_1 \oplus Sq_1 \oplus y_1) \oplus (A_1 \times B_2 \oplus z_2) \quad (5)$$
$$C_2 = (A_2 \times B_0 \oplus z_1 \oplus z_3) \oplus (A_2 \times B_1 \oplus z_2) \oplus (A_2 \times B_2 \oplus Sq_2 \oplus y_0) \quad (6)$$

Note that this multiplier does not support dependent inputs, but independent inputs are obtained by storing the output of the linear map in a register. In the original design, the square scaler terms (Sq_0, Sq_1, Sq_2) were added to the output of the Stage 1 DOM-*dep* multiplier in the second pipeline stage. This can potentially cause a related input sharing to the multiplier in Stage 2 due to glitches. Therefore, we perform the addition of these terms already in Stage 1 by adding them to the inner-domain multiplication terms before the register layer. As a nice benefit, this saves registers to store the square scaler output in the original design.

Another issue is that the Stage 1 multiplier might temporarily only output the same-domain terms due to glitches if, e.g., the wire length of cross-domain terms is significantly longer. In that case, the Stage 2 multiplier, which multiplies the lower and higher two bits of the Stage 1 multiplier, might temporarily operate on related inputs. Therefore, we use 2×4 random bits y_0 and y_1 to also refresh the inner-domain terms. In order to maintain second-order probing security, the cross-domain terms need to be refreshed with an additional z_3 in this case. Otherwise, an attacker can place a probe in the calculation phase of the Stage 2 multiplier to get a combination of masks, which is used to protect the integration phase of the Stage 1 multiplier.

Multiplier in Stage 2 (Type C). We want to replace the DOM-*dep* multipliers in Stage 3 by a DOM-*indep* multiplier. The DOM-*indep* multiplier in Stage 3 only supports independent inputs, so the DOM-*indep* multiplier in this stage needs

Table 1.: Comparison of the amount of fresh randomness required for the insecure and fixed second-order DOM-*dep* multipliers, and the resulting insecure, fixed and optimized second-order DOM AES S-boxes. For the S-box constructions we give in brackets the amount of required random bits per stage.

Construction		Fresh randomness	Area
Insecure second-order DOM-*dep* [GMK16]	$GF(2^2)$	12 bit	N/A
	$GF(2^4)$	24 bit	
Fixed second-order DOM-*dep*	$GF(2^2)$	16 bit	
	$GF(2^4)$	32 bit	
Insecure second-order DOM AES S-box [GMK16]		84 bit (24/12/24/24)	N/A
Fixed second-order DOM AES S-box		104 bit (32/16/32/24)	4.37 kGE
Optimized second-order DOM AES S-box		78 bit (24/18/12/24)	4.29 kGE

to be modified such that it generates an independent output sharing. To do so, we need to perform changes similar to Stage 1, including shifting the addition of square scaler terms and additional protection for inner-domain and cross-domain terms. In summary, the modified DOM-*indep* multiplier, which will be referred to as the Type C multiplier, used in Stage 2, with parenthesis indicating registers, is given by:

$$C_0 = (A_0 \times B_0 \oplus Sq_0 \oplus y_0 \oplus y_1) \oplus (A_0 \times B_1 \oplus z_0 \oplus z_3) \oplus (A_0 \times B_2 \oplus z_1 \oplus z_5) \tag{7}$$

$$C_1 = (A_1 \times B_0 \oplus z_0 \oplus z_4) \oplus (A_1 \times B_1 \oplus Sq_1 \oplus y_1 \oplus y_2) \oplus (A_1 \times B_2 \oplus z_2 \oplus z_5) \tag{8}$$

$$C_2 = (A_2 \times B_0 \oplus z_1 \oplus z_3) \oplus (A_2 \times B_1 \oplus z_2 \oplus z_4) \oplus (A_2 \times B_2 \oplus Sq_2 \oplus y_0 \oplus y_2) \tag{9}$$

Note that, also this multiplier does not support dependent inputs, but independent inputs are obtained by appropriate refreshing in the stage before. Compared to the multiplier in Stage 1 (Type B), we need more randomness for refreshing the multiplication terms. In total, 3×2 bits are needed for inner-domain terms (y_0, y_1, y_2), and 6×2 bits are needed for cross-domain terms $(z_0, z_1, z_2, z_3, z_4, z_5)$.

Multipliers in stages 3 and 4 (Type A). After performing these changes, the DOM-*dep* multiplier in Stage 3 can simply be replaced by the original DOM-*indep* multiplier (Type A) because independent inputs are obtained by refreshing in Stage 2. The multiplier in Stage 4 has originally been a DOM-*indep* multiplier and therefore, no further modifications are required there.

3.4. Discussion

In Table 1 we compare the randomness properties of the different constructions. As stated by [GMK16], it requires 6/12 bits of fresh randomness for a $GF(2^2)/GF(2^4)$

DOM-*indep* multiplier. The insecure DOM-*dep* multiplier requires 12/24 bits of fresh randomness for $GF(2^2)/GF(2^4)$. The fixed version of the DOM-*dep* multiplier, which works with our fix, requires 16/32 bits of fresh randomness. The amount of 84 bits for the whole insecure S-box denotes to 24 bits in Stage 1, 12 bits in Stage 2, $2 \times 12 = 24$ bits in Stage 3, and 2×12 bits in Stage 4. When exchanging the DOM-*dep* multipliers in that design with our fixed multipliers, the final construction leads to a randomness consumption of 104 bits, implying an increase of 24%. More precisely, 32 bits of fresh randomness are now needed in Stage 1, 16 bits in Stage 2, 32 bits in Stage 3 and 24 bits in Stage 4. Our optimized second-order S-box design, which does not use any DOM-*dep* multipliers, has a lower randomness consumption of 78 bits and also a slightly lower area (4.29 kGE) compared to the originally proposed version.

4. COTG-based Design of AES

Using the S-box design described in Section 3 directly in a masked AES implementation requires 15 600 bits of fresh randomness per encryption. In this section, we show how a COTG-based concept inspired by [Dae17] can be used to reduce this number to only 3 200. In general, each S-box requires 78 bits for refreshing the multiplication terms in the multipliers. Our main goal is to replace as many of these 78 bits by *guards*, i.e., shares of state bytes of another unrelated S-box, and use fresh randomness produced by an RNG where necessary, such that in total, connecting an RNG producing 64 bits of fresh randomness per cycle to the design is sufficient.

We give a general overview of our concept in Section 4.1. In Section 4.2, we give more details on the exact COTG-based SubBytes operation for the shared plaintext. In Section 4.3, we show how a similar concept applies to the key schedule. We verify the basic assumptions made for our concept with Coco, as described in more detail in Section 6.

4.1. Overview

The AES round function can be divided into four smaller *super boxes*, mapping a 32-bit input to a 32-bit output by applying SubBytes, MixColumns, AddRoundKey, and the second SubBytes function. The four input bytes of a super box are the columns of the state when viewed after ShiftRows. From a masking point of view, the non-linear SubBytes operation processes each state byte individually but combines the share domains. In contrast, the linear MixColumns operation combines the four state bytes of a super box but does this for each share domain individually.

These considerations suggest some general constraints regarding a COTG-based AES design. First, without COTG, the state bytes are kept isolated from each other until MixColumns, while with COTG, other state bytes are mixed in during

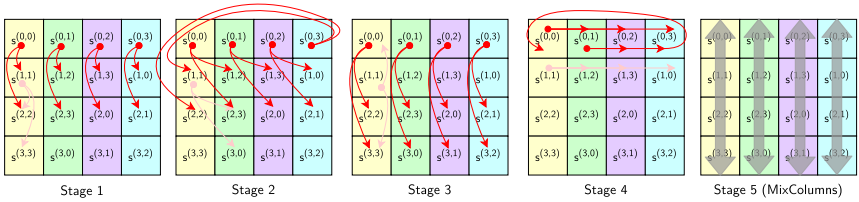


Figure 2.: Overview of the proposed COTG concept. The squares represent the 4x4 AES state grouped in four super boxes (=the state after ShiftRows). For a specific state byte (indicated by •), the red arrow illustrates the other state bytes used as guards. In the last stage, we sketch the MixColumns operation combining all bytes of a super box.

the SubBytes operation in terms of randomness required by the multipliers. As noted by [BDZ20], this could change the diffusion properties of the masked cipher in an unfavorable way, for which we account with super box-wise resharing using fresh randomness before MixColumns similar to [DSM22]. Second, on the level of a single S-box, we need to choose guards for refreshing the multipliers such that they are always independent of the multipliers' inputs. This becomes even more complex considering that a multiplier input is usually just the output of another multiplier from the previous stage, which again directly relates to the guards used there.

Therefore, from the view of a single S-box (located in super box i) in our design we make the following decisions regarding which other state bytes can be used as guards for refreshing (we sketch this in Figure 2):

- MixColumns combines all state bytes of a super box, i.e., all guards used in all Stage 4 multipliers of the super box bytes are combined. Therefore, the guards need to be chosen from the three foreign super boxes $i+1, i+2, i+3$. To avoid changing diffusion properties, we refresh the inner-domain terms with fresh randomness.
- Taking the guards for Stage 4 from the three foreign super boxes leaves us with no choice but to ensure that the multiplier inputs are related to the domestic super box. The inputs are (a) the plain shares after the linear map (by default related to domestic super box i) and (b) the output of the Stage 3 multiplier. By choosing guards from the domestic super box i in combination with fresh randomness, we get independence here as well. In order to obtain the independence even in the presence of glitches, the inner-domain terms in Stage 3 are again refreshed with fresh randomness.
- The inputs of the Stage 3 multiplier are the outputs of stages 1 and 2. However, the guards of the Stage 3 multiplier are independent of any unmasked state byte because they are combined with fresh randomness.

Table 2.: Assignment of guards and fresh randomness to refresh the inner- and cross-domain terms of the DOM-*indep* multipliers in our design. The operator $X[a:b]$ extracts the bits in range from b to (including) a from a given binary word X. The 64 bits of fresh randomness R given to the design in every cycle is arranged in rows $R0, R1, R2, R3$ of 16 bits each.

	DOM-*indep* multiplier					
	1	**2**	**3/1**	**3/2**	**4/1**	**4/2**
z_0	$s_0^{(i+1,j+1)}$	Ri[7:0]	$s_2^{(i+2,j+2)}[5:0]$ $\oplus$ Ri[5:0]	$s_0^{(i+3,j+3)}[5:0]$ $\oplus$ Ri[13:8]	$s_0^{(i,j+1)}[3:0]$	$s_1^{(i,j+2)}[7:4]$
z_1					$s_0^{(i,j+1)}[7:4]$	$s_2^{(i,j+3)}[3:0]$
z_2	Ri[7:0]				$s_1^{(i,j+2)}[3:0]$	$s_2^{(i,j+3)}[7:4]$
z_3			-	-	-	-
z_4	$s_1^{(i+2,j+2)}$ $\oplus$ Ri[15:8]	$s_0^{(i+1,j+2)}$ $\oplus$ Ri[15:8]	-	-	-	-
z_5			-	-	-	-
y_0	-		Ri[7:6]	Ri[7:6]	Ri[3:0]	Ri[11:8]
y_1	-		Ri[15:14]	Ri[15:14]	Ri[7:4]	Ri[15:12]
y_2	-	$s_1^{(i+2,j+3)}[1:0]$	-	-	-	-

Hence, we can simply choose guards from the domestic super box for Stage 1 and guards from the neighbor super box for stage 2.

4.2. COTG for SubBytes of Plaintext

Choice of guards for Stage 4. Stage 5 of our design computes the complete linear layer, i.e., the inverse linear map, ShiftRows, MixColumns, AddRoundKey, and the linear map of SubBytes of the next round. Each operation is applied exactly once per share and does not combine shares of different domains. The linear mappings of the S-box mix the bits of a share byte, and AddRoundKey combines the state bytes bitwise with unrelated key material. MixColumns however combines the bytes of each super box in the design, or, when viewed from a masking perspective, combines the refreshed multiplication terms of the Stage 4 multipliers of the four super box bytes. Due to glitches, every masked multiplication term can be observed individually, and thus, all their combinations. In order to refresh these multiplication terms, which is done in the two DOM-*indep* multipliers using z_0, z_1, and z_2, we instantiate 24 bits of guards. As shown in Table 2, we use *guards of three different foreign super boxes with rotating share domains* for this purpose. For example, the Stage 4 multipliers of the first two super boxes use the following state bytes as guards:

$$s^{(0,0)} : s_0^{(0,1)}, s_1^{(0,2)}, s_2^{(0,3)} \qquad s^{(0,1)} : s_0^{(0,2)}, s_1^{(0,3)}, s_2^{(0,0)}$$

$$s^{(1,1)} : s_0^{(1,2)}, s_1^{(1,3)}, s_2^{(1,0)} \qquad s^{(1,2)} : s_0^{(1,3)}, s_1^{(1,0)}, s_2^{(1,1)}$$

$$s^{(2,2)} : s_0^{(2,3)}, s_1^{(2,0)}, s_2^{(2,1)} \qquad s^{(2,3)} : s_0^{(2,0)}, s_1^{(2,1)}, s_2^{(2,2)}$$

$$s^{(3,3)} : s_0^{(3,0)}, s_1^{(3,1)}, s_2^{(3,2)} \qquad s^{(3,0)} : s_0^{(3,1)}, s_1^{(3,2)}, s_2^{(3,3)}$$

Rotating share domains means that we use share domain 0 for the first guard, share domain 1 for the second, and share domain 2 for the third. We cannot use the same guard domain, e.g., domain 0, for all guards because that would lead to many Stage 4 multiplication terms being refreshed with the same guard. By rotating the domains, every state byte share is used exactly once in the Stage 4 multipliers. Assume that the guards for an S-box are not distributed across super boxes, but that for super box i, we use state bytes of the same domestic super box i. That implies that $s^{(0,0)}$ uses $s_2^{(3,3)}$, $s^{(1,1)}$ uses $s_1^{(3,3)}$ and $s^{(2,2)}$ uses $s_0^{(3,3)}$ as a guard, and hence, in MixColumns, state byte $s^{(3,3)}$ is unmasked. The same holds when super box i uses state bytes of the same foreign super box. Therefore, the guards need to originate from *three different foreign superboxes*. At the same time, it is important to note that every MixColumns operation combines shares of exactly one share domain of each super box. For example, super box 0 uses guards from super box 1, but all of share domain 0. That is important to prevent an attacker from placing two probes in the MixColumns operations of different super boxes. Additionally, we use the 64 bits of fresh randomness produced by the RNG to refresh the inner-domain terms with y_0 and y_1.

Similar to [DSM22], instead of refreshing the complete state (which would require 256 bits of fresh randomness), we align the 64 bits of fresh randomness into four rows $R0, R1, R2, R3$ of 16 bits each such that the randomness is reused in every super box.

Choice of guards for Stage 3. The Stage 4 DOM-*indep* multipliers multiply (a) the plain input shares of the S-box after the linear map, with (b) the output of the Stage 3 multipliers. The guards used in Stage 4 must be independent of both (a) and (b). In the case of (a), independence between the plain input shares of a specific S-box and state bytes of other super boxes is naturally given. In the case of (b), the independence is determined by the output of the multipliers in Stage 3 and, therefore, by the guards used in Stage 3. In Stage 3, $2 \times 6 = 12$ bits are required for refreshing cross-domain multiplication terms (z_0, z_1, z_2 in multipliers 3/1 and 3/2), and additionally, 2×4 bit are required for refreshing inner-domain multiplication terms (y_0, y_1) to achieve that even in the presence of glitches, the inputs to Stage 4 are independent. In total, this makes 20 bits, which can however be reduced to 16 bits because, in the multiplier 3/1 and the multiplier 3/2, the same values for y_0 and y_1 can be used.

In summary, we therefore need to come up with 16 bits of randomness per S-box. Similar to Stage 4, we again arrange the 64 bits of fresh randomness generated in this cycle by the RNG in four rows of 16 bits and re-use this randomness in every super box. Verification with Coco reveals that while this is valid for inner-domain terms (y_0, y_1), the cross-domain terms must be refreshed with unique randomness (z_0, z_1, z_2).As shown in Table 2, we use a trick to generate unique terms by combining the fresh randomness from the RNG with guards taken from the domestic super box. For example, the multiplier 3/1 of the first

two super boxes uses the following values for z_0, z_1, z_2:

$$s^{(0,0)} : s_2^{(2,2)}[5{:}0] \oplus R0[5{:}0] \qquad s^{(0,1)} : s_2^{(2,3)}[5{:}0] \oplus R0[5{:}0]$$
$$s^{(1,1)} : s_2^{(0,0)}[5{:}0] \oplus R1[5{:}0] \qquad s^{(1,2)} : s_2^{(3,0)}[5{:}0] \oplus R1[5{:}0]$$
$$s^{(2,2)} : s_2^{(1,1)}[5{:}0] \oplus R2[5{:}0] \qquad s^{(2,3)} : s_2^{(0,1)}[5{:}0] \oplus R2[5{:}0]$$
$$s^{(3,3)} : s_2^{(2,2)}[5{:}0] \oplus R3[5{:}0] \qquad s^{(3,0)} : s_2^{(1,2)}[5{:}0] \oplus R3[5{:}0]$$

By doing so, the uniqueness of the term is given by Ri within the super box, and by the guards across super boxes, and every Stage 3 multiplier in all S-boxes uses unique values to refresh the multiplication terms. Similar to Stage 4, we perform share domain rotation by using share 2 for the 3/1 multipliers and share 0 for the 3/2 multipliers in order to achieve that within a super box, two different shares of a state byte are used as guards.

Choice of guards for Stage 2. The Stage 3 DOM-*indep* multipliers multiply the output of the Stage 2 multiplier with the output of the Stage 1 multiplier. The guards used in Stage 3 are inherently independent of these because the randomness generated by the RNG in Stage 3, which is used to *mask* the guards, is only used in that cycle. Therefore, the choice of guards for stages 1 and 2 is relatively unconstrained as long as they are independent of each other (otherwise, a DOM-*dep* multiplier would need to be used). In Stage 2, 18 bits are required for refreshing cross-domain multiplication terms $(z_0, z_1, z_2, z_3, z_4, z_5)$ and inner-domain multiplication terms (y_0, y_1, y_2). Using an analysis with Coco, we find out that for second-order probing security, z_0, z_1, z_2, z_3 can be re-used across super boxes, while the rest of the values need to be unique. As shown in Table 2, we apply a similar trick as in Stage 3 to generate this uniqueness: We use the fresh randomness generated by the RNG, distribute it over the columns of the state, and re-mask it with guards as necessary to obtain a unique random value. For example, the values used for refreshing in the Stage 2 multipliers are:

$$s^{(0,0)} : R0[7{:}0], s_0^{(1,2)} \oplus R0[15{:}8], s_1^{(2,3)}[1{:}0] \qquad s^{(0,1)} : R0[7{:}0], s_0^{(1,3)} \oplus R0[15{:}8], s_1^{(2,0)}[1{:}0]$$
$$s^{(1,1)} : R1[7{:}0], s_0^{(2,3)} \oplus R1[15{:}8], s_1^{(3,0)}[1{:}0] \qquad s^{(1,2)} : R1[7{:}0], s_0^{(2,0)} \oplus R1[15{:}8], s_1^{(3,1)}[1{:}0]$$
$$s^{(2,2)} : R2[7{:}0], s_0^{(3,0)} \oplus R2[15{:}8], s_1^{(0,1)}[1{:}0] \qquad s^{(2,3)} : R2[7{:}0], s_0^{(3,1)} \oplus R2[15{:}8], s_1^{(0,2)}[1{:}0]$$
$$s^{(3,3)} : R3[7{:}0], s_0^{(0,1)} \oplus R3[15{:}8], s_1^{(1,2)}[1{:}0] \qquad s^{(3,0)} : R3[7{:}0], s_0^{(0,2)} \oplus R3[15{:}8], s_1^{(1,3)}[1{:}0]$$

Note that we again perform share domain rotation, i.e., every byte in a superbox uses guards from two different domains.

Choice of guards for Stage 1. The Stage 2 DOM-*indep* multiplier multiplies the four most significant bits of the Stage 1 multiplier output with the four least significant bits. The 24 bits required for refreshing in the Stage 1 multiplier hence need to be chosen independently of the guards in Stage 2. Analysis with Coco reveals that in this situation, the values for z_0, z_1, z_4 and z_5 need to be unique, while z_2 and z_3 can again be re-used across super boxes. As shown in Table 2, for z_2 and z_3 we use a byte of fresh randomness from the RNG, which is re-used once per super box. For z_4 and z_5 we use another byte of fresh randomness from the RNG, which is also re-used once per super box, but made unique by re-masking with a guard from the domestic super box. For z_0 and z_1 we need a unique value as well, however, the 16 bits of randomness available are already used up, and therefore, we directly use as a guard a state byte from the same domestic super box. For example, the values used for refreshing in the Stage 1 multipliers are:

$s^{(0,0)} : s_0^{(1,1)}, R0[7{:}0], s_1^{(2,2)} \oplus R0[15{:}8]$ $\quad$ $s^{(0,1)} : s_0^{(1,2)}, R0[7{:}0], s_1^{(2,3)} \oplus R0[15{:}8]$

$s^{(1,1)} : s_0^{(2,2)}, R1[7{:}0], s_1^{(3,3)} \oplus R1[15{:}8]$ $\quad$ $s^{(1,2)} : s_0^{(2,3)}, R1[7{:}0], s_1^{(3,0)} \oplus R1[15{:}8]$

$s^{(2,2)} : s_0^{(3,3)}, R2[7{:}0], s_1^{(0,0)} \oplus R2[15{:}8]$ $\quad$ $s^{(2,3)} : s_0^{(3,0)}, R2[7{:}0], s_1^{(0,1)} \oplus R2[15{:}8]$

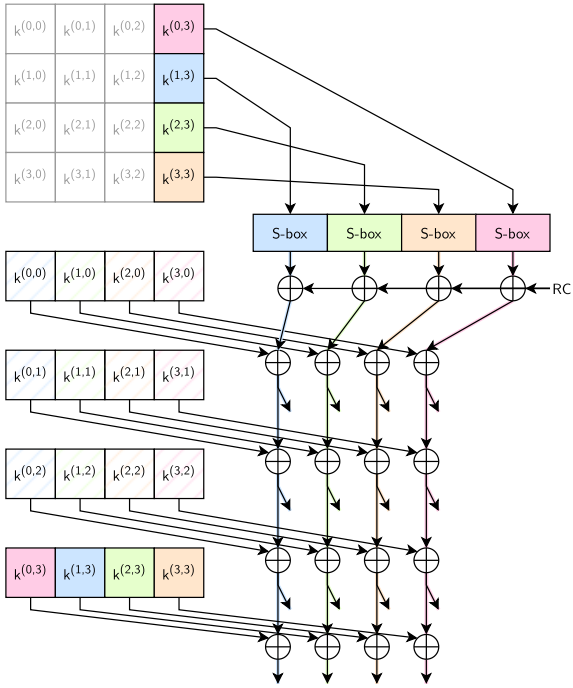
$s^{(3,3)} : s_0^{(0,0)}, R3[7{:}0], s_1^{(1,1)} \oplus R3[15{:}8]$ $\quad$ $s^{(3,0)} : s_0^{(0,1)}, R3[7{:}0], s_1^{(1,2)} \oplus R3[15{:}8]$

4.3. COTG for SubWord of Key Schedule

In our AES design, we use the same shared S-box design for the key as for the plaintext. Masking the key schedule using COTG is however much simpler than for the plaintext because only four key bytes are transformed using SubWord, which is comprised of four S-boxes, and no MixColumns operation is performed during the key schedule (c.f. Figure 3). Therefore, we first identify key state bytes that cannot be used in a straightforward way as guards in the SubWord operation of the key schedule, that are, the set of key bytes combined with each SubWord input byte. This set includes the SubWord input bytes $k^{(0,3)}, k^{(1,3)}, k^{(2,3)}, k^{3,3}$ themselves, and then for each byte, the three other key bytes added to the S-box output later in the key schedule. For example, for $k^{(0,3)}$ we do not use $k^{(3,0)}, k^{(3,1)}, k^{(3,2)}$ as guards. In Figure 3, we mark the key bytes not used as guards for a specific S-box with stripes of the respective color.

For each of the four input bytes, we can then simply assign the remaining key state bytes as guards for the respective S-box and perform share-domain rotation on that. Using this technique, we can obtain the second-order probing security of the construction. An adversary placing two probes in the same S-box of the key schedule cannot probe a complete sharing of a guard byte because per S-box, at most one share of a guard is used. With two probes in two different S-boxes, an adversary can therefore at most probe two out of three shares.

The RNG connected to the AES design produces 64 bits of fresh randomness



(a)

$k^{(0,3)}$: $k_0^{(0,0)}, k_1^{(0,1)}, k_2^{(0,2)}, k_1^{(1,0)}, k_2^{(1,1)}, k_0^{(1,2)}, k_2^{(2,0)}, k_0^{(2,1)}, k_1^{(2,2)}, R[15:0]$

$k^{(1,3)}$: $k_0^{(1,0)}, k_1^{(1,1)}, k_2^{(1,2)}, k_1^{(2,0)}, k_2^{(2,1)}, k_0^{(2,2)}, k_2^{(3,0)}, k_0^{(3,1)}, k_1^{(3,2)}, R[31:16]$

$k^{(2,3)}$: $k_0^{(2,0)}, k_1^{(2,1)}, k_2^{(2,2)}, k_1^{(3,0)}, k_2^{(3,1)}, k_0^{(3,2)}, k_2^{(0,0)}, k_0^{(0,1)}, k_1^{(0,2)}, R[47:32]$

$k^{(3,3)}$: $k_0^{(3,0)}, k_1^{(3,1)}, k_2^{(3,2)}, k_1^{(0,0)}, k_2^{(0,1)}, k_0^{(0,2)}, k_2^{(1,0)}, k_0^{(1,1)}, k_1^{(1,2)}, R[63:48]$

(b)

Figure 3.: (a) The AES key schedule. We mark the input bytes of SubWord with colors, and hatch the key state bytes which are later combined with a specific input byte.
(b) The assignment of guards for the S-boxes of the key schedule.

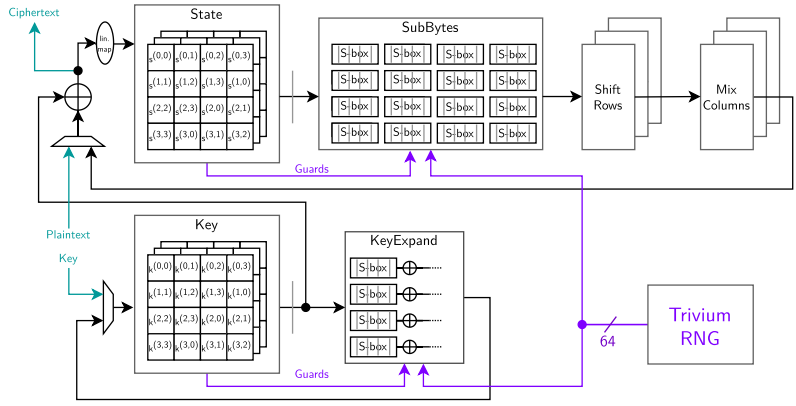


Figure 4.: Architecture of our second-order AES implementation. Pipeline stages are sketched with gray lines, inputs and outputs are marked in turquoise, and terms used for refreshing the S-box multipliers (guards and fresh randomness) are printed in purple.

per cycle for encrypting the plaintext. However, in Stage 5 of computing the S-box for the plaintext, no fresh randomness is required because only linear operations are performed, and we can use the 64 bits of fresh randomness produced in that cycle for refreshing the key schedule. We distribute the 64 bits over the four S-boxes, such that we add 16 distinct bits per S-box. By that, we can keep the refreshing of plaintext and key completely independent of each other, which is also important for probing security across multiple rounds, as discussed in Section 6.

5. Architecture

Masked AES hardware implementations either follow a *serial* or a *parallel* design paradigm. *Serial* AES designs instantiate the S-box once, which is fed with a new state or key byte every clock cycle. Most existing masked AES designs in literature focus on serial designs, including [Ask+22; Bil+14; Bil+15; DSM22; GMK16; Mor+11; Sug19], which is suitable for low-area, low-power purposes, but less for high throughput or low latency [Uen+16]. *Super box-serial* designs instantiate four S-boxes that are fed with a new super box every clock cycle and therefore provide a higher performance at the cost of area. *Parallel* or *round-based* AES designs instantiate the S-box 20 times, 16 inside SubBytes and 4 inside KeyExpand, which enables even higher performance at the cost of area. Our design follows a parallel architecture, as we use the AES implementation of the OpenTitan project as a basis. OpenTitan includes a first-order masked AES

with a fully-parallel data path in order to achieve higher performance, but also because parallel architectures increase the noise in a system, which makes SCA harder [low23].

We give a sketch of our design in Figure 4. It takes 50+1 cycles to encrypt a block of 16 plaintext bytes. One cycle in the beginning is needed because the key schedule is started 1 cycle earlier than the processing of the plaintext in our design, such that the round key used in AddRoundKey for a specific round always comes from the key state registers. The linear map of our S-box design is now computed in the fifth stage of a round, which means the state registers of our implementation do not store the plain AES state but the state in the normal basis. We connect a Trivium RNG [Can06] to our design in order to further analyze the area overhead caused by utilizing multiple RNGs. We choose Trivium only as an example that can, in practice, be replaced by any other RNG producing randomness at a sufficient quality. Our Trivium implementation provides 64 bits of fresh randomness per clock cycle. The randomness produced in the first four cycles of a round is consumed by the plaintext encryption (256 bits), and the randomness produced in the fifth cycle is consumed by the key schedule (64 bits). Our design requires 320 bits of fresh randomness per round, or 3 200 bits for 10 rounds.

5.1. Implementation and Comparison

We implement our design and obtain area measures using Cadence Genus Synthesis Solution 19.11-s087_1 for synthesis. All data is collected for a UMC 64 nm process and is expressed in 2-input NAND gate equivalents. The area of one NAND gate is 1.44 μm^2. In Table 3a, we give details about the area consumption of our AES design, which is in total 102 kGE. Two-thirds of the total area is attributed to the S-box instances for the plaintext/data, followed by the S-box instances for the key schedule. Since, to the best of our knowledge, our design is currently the only second-order parallel AES design, any direct comparison on cipher-level to related work is not possible. [Ask+22] provide a first-order parallel AES design with a 5-cycle S-box requiring 102.4 kGE, which is about the same as our second-order design. However, the comparison is not fair because the design does not use any online randomness at all, and the gate libraries as well as design compilers do not match.

On S-box level, we compare our design to related work in literature, as shown in Table 3b. However, it must be noted that these implementations use different CMOS libraries and design compilers, and therefore, the comparison only serves as a rough point of reference. Our optimized S-box design requires 4.3 kGE, which is slightly less (-0.1 kGE) than the fixed version of [GMK16], in which we include the fixed DOM-*dep* multipliers. Compared to the original versions of [GMK16], the area consumption of our design has not changed significantly. [Sim+22] and [Nag+22] propose S-box designs with a much lower latency than ours (1 cycle) but also with a higher area consumption. Gross et al. [GIB18]

Table 3.: Evaluation and comparison of our design in terms of area (* including control logic for COTG)

Module	**Area**	
	[%]	*[kGE]*
DOM-AES with COTG		
Data SubBytes	62%	63.7
Key SubWord	15%	15.7
MixColumns	3%	2.8
Control logic, state registers, etc	20%	19.8
Total AES	100%	**102**

(a)

2nd-order AES S-box	**Area [kGE]**	**Latency [cycles]**	**Rand. [bits]**	**CMOS library**
[Cnu+15]	7.8	6	126	NanGate 45nm
[Cnu+16]	3.8	5	162	NanGate 45nm
[GIB18]	57.1	2	4446	UMC 90nm
[Nag+22]	14.8	1	51	UMC 65nm
[Sim+22]	11.4	1	108	N/A (40nm)
[GMK16]	5.3	8	54	UMC 180nm
[GMK16] (insecure)	5.7	5	84	UMC 180nm
[GMK16] (fixed)	4.4	5	104	UMC 65nm
This work	**4.3**	**5**	**78**	UMC 65nm

(b)

Module	**Area**	
	[%]	*[kGE]*
DOM-AES with COTG, 1 Trivium instance		
AES*	87%	102
Trivium instance	5%	5.2
Outer control logic	8%	9.4
Total	100%	**116.6**
DOM-AES without COTG, 7.5 Trivium instances		
AES	68%	96.9
Trivium instances	25%	35.7
Outer control logic	7%	10.4
Total	100%	**142.1**
DOM-AES with fixed DOM-*dep*, 10 Trivium instances		
AES	65%	115.1
Trivium instances	30%	51.7
Outer control logic	5%	9.4
Total	100%	**176.2**

(c)

construct another DOM-S-box design focused on low-latency (2 cycles) without dual-rail logic, which however has a higher overhead in area and randomness than our design. The five-cycle S-box proposed by [Cnu+16] has a slightly lower area than our design but requires more than twice as much randomness.

In Table 3c, we compare our design with COTG to two versions of the design without COTG, connected to multiple instances of the Trivium RNG. This comparison highlights how important the reduction of randomness in a masked design is to achieve area efficiency. We evaluate our DOM-AES design using COTG, to which we connect a single Trivium instance, providing 64 bits of fresh randomness per clock cycle. The whole design requires 116.6 kGE, and the RNG makes 5% of the total area. We compare this to a version of our design where we do not use COTG but exclusively use fresh randomness for refreshing in the S-boxes, which consequently requires 7.5 Trivium instances. The total design area is 142.1 kGE, thus, represents an overhead of 22%. In a third scenario, we analyze the area consumption of the original DOM-AES design using our fixed DOM-*dep* multipliers. Here, 10 Trivium instances necessary, which consume 30% of the total design area, which is 176.2 kGE and represents an overhead of about 50% compared to our design using COTG. The area of the AES core has an overhead of 13% by using the DOM-*dep* multipliers instead of the smaller DOM-*indep* multipliers. Note that our AES design provides plenty of further possibilities for optimization, which would eventually reduce the area even more, including the elimination of the extensively used control logic for COTG. Additionally, instead of placing multiple Trivium instances, the Trivium state update function can further be unrolled to save area, as described in [Cas+23].

5.2. Application to other use-cases

Despite our decision to follow a parallel (round-based) design concept, the proposed concept for COTG can easily be carried over to serial and super box-serial architectures. The choice of guards stays the same; only the distribution of the randomness supplied by the RNG slightly changes. In a parallel design, all four super boxes are computationally in the same pipeline stage p in a specific cycle, and the 64 bits of fresh randomness are sent to that stage. In a super box-serial design, super box 0 would be in stage p, but super box 1 would be in stage $p-1$. Hence, one can send the 64 bits of fresh randomness to stage p for super box 0 and to stage $p-1$ for super box 1. Similar considerations are possible for a serial design, although an RNG supplying less than 64 bits would be sufficient.

While we focus on the second-order case, the proposed techniques can theoretically also be applied to higher-order ($d > 2$) DOM-protected AES implementations. To do so, one needs first to replace the DOM-*dep* multipliers in the S-box with DOM-*indep* multipliers, which requires adding even more fresh randomness per DOM-*indep* multiplier. Next, independent state bytes need to be identified, which can be used as guards in each S-box stage, similar to what is

done in this work. We expect that this analysis, which is not trivial and becomes harder the higher the masking order, needs to be done individually for every order, while some knowledge, e.g., about the general dependency of state bytes, can be re-used from the second-order case.

The applicability of the concept to other ciphers, potentially protected by techniques other than DOM, highly depends on the concrete construction and requires a more in-depth individual analysis. For example, we expect that a similar technique can be applied to Ascon [Dob+21], and obtaining a COTG-based concept might be even less complex since DOM-masked Ascon implementations are available without using DOM-*dep* multipliers [GM17].

6. Security Evaluation

In this section, we elaborate on the security of our second-order DOM-AES implementation using COTG. First, we provide a formal security analysis of the design for which we use the formal verification tool Coco [Gig+21]. Second, we provide a practical security analysis by porting the circuit to an FPGA and showing that no leakage could be detected using TVLA with up to 100 million traces.

6.1. Formal verification setup

In this work, we use Coco [Gig+21] for formally verifying our design in the time-constrained probing model. The original purpose of Coco is to verify masked software implementations directly on the CPU netlist by incorporating control signals originating from the software execution. Given that Coco operates on gate-level netlists, it can also be used directly to verify masked hardware circuits with control logic, as demonstrated in [HB21]. To apply Coco, our design is first synthesized with Yosys [Wol16] to obtain such a gate-level netlist. We simulate the design to obtain values for control signals generated by the state machine in our design for the verification. Additionally, labels are assigned to the circuit inputs in order to indicate their purpose (share of a sensitive variable, fresh randomness, or unimportant/control signal). We further add some small modifications to Coco for our needs. For example, the original version of Coco constructs one SAT equation per sensitive bit in the circuit and then uses the incremental CaDiCaL SAT solver [Bie+20] to solve the equations in a sequential order. More precisely, the solver first checks the equation of the first sensitive bit and then uses the learned clauses for the remaining ones. Incremental SAT solving however comes with a certain overhead, e.g., for storing the learned clauses, and we found out that for our second-order hardware designs, the amount of re-usable learned clauses is so small that incremental solving does not pay off. Therefore, we use a parallel solver that solves all SAT equations individually but at the same time in parallel. We therefore adapt the Coco backend such that it

uses the Kissat [Bie+20] solver. All experiments are executed on a machine with 88 CPU cores with 500 GB of RAM, such that approximately one CPU core is available per SAT formula.

6.2. Formal security of the design

In order to evaluate the security of our design, we follow a multi-step approach. First, we formally verify the second-order security of the S-box, treating the 78 input bits for refreshing the multipliers as fresh randomness first. Second, we take a look at the security of the design for one round on super box-level, including the usage of guards for refreshing, and formally verify it for both the key schedule and plaintext using Coco. Finally, we comment on the situation for the later rounds.

Formal verification of the S-box. As a first step, we formally verify with Coco that our proposed fix for the second-order DOM-*dep* multipliers is secure. For that, we create a $GF(2^2)$ and a $GF(2^4)$ DOM-*dep* multiplier implementation in System Verilog and verify the security in the time-constrained probing model for both implementations, which takes a few seconds. We then focus on the S-box construction proposed in Section 3.3, which does however not use the fixed DOM-*dep* multipliers to save randomness, which we verify for six cycles. We mark the three input shares (eight bits each) as sensitive values and the 78 bits of randomness for refreshing, which we all mark as uniformly random. Coco confirms the second-order security of our S-box implementation in the time-constrained probing model after running for approximately 1.5 days.

Formal verification of COTG for SubWord of key schedule. In order to formally verify one round of the key schedule using COTG, we label the three shares of the complete 128-bit key state as sensitive variables. During the computation of SubWord, these will be used as guards for refreshing. Additionally, we mark the 64 bits of fresh randomness required by the key schedule in Stage 4 of the S-boxes. With Coco, we can confirm the probing security of the construction computing four S-boxes in parallel over one round in 2 days and 18 h. This involves solving one SAT formula per unshared key bit, i.e., 128 SAT formulas in parallel. Not every SAT formula needs the same amount of time to solve, for example, the formulas of key bits that are not processed by SubWord are solved very quickly (in 2 s), while it takes up to the indicated 2 days and 18 h to check the security of key bits processed by the S-box.

One of the goals when constructing our design was to keep the refreshing terms used in the key schedule and plaintext isolated from each other to allow for easier security analysis. That is, no randomness or guards for refreshing are used in both the key schedule and the processing of the plaintext, and the only meeting point is AddRoundKey. Processing of the plaintext does not require fresh randomness

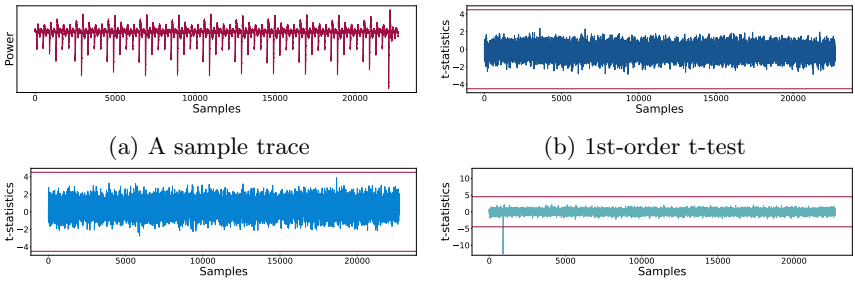
in Stage 5 where the linear operations are done, but still, the RNG produces 64 bits of fresh randomness in that cycle, which we use for refreshing the key state after SubWords, impeding to probe key bytes in two different rounds.

Formal verification of COTG for SubBytes of plaintext. Compared to verification of the key schedule, verification of the COTG-based concept for the plaintext is much harder due to more complex dependencies between the state bytes. First, 16 S-boxes are computed in parallel instead of only four, and the guards used for these S-boxes are at the same time sent through their own S-box, where other guards are used. Second, we are using a combination of guards and fresh randomness for refreshing the multipliers connected by the $\oplus$ operation. Due to these two aspects, verifying a complete round for the complete 128-bit state becomes computationally infeasible.

Therefore, we constrain the verification to super boxes 0 and the first byte of super box 1 ($s^{(0,1)}$), i.e., we mark the whole 128-bit state of the AES as sensitive but disable the S-box computation for ($s^{(1,2)}, s^{(2,3)}, s^{(3,0)}$) and the bytes of the super boxes 2 and 3. This should not affect the verification of super box 0 since, in the first three stages, every super box uses guards only from the same or neighbor super box. Using this setup, we verify the construction for the first three stages, including the resharing phase of Stage 4. In Section 4.2, we discuss that inputs to Stage 4 are independent of each other, which allows to start the verification after Stage 3, assuming independent input shares. We verify the design beginning with the integration phase in Stage 3 until the end of Stage 5, including MixColumns, which is completed successfully.

An attempt to verify a complete round at once was not conclusive, as the verification has been running for 55 days, and no leak has been found yet, but the security for all bits could not be confirmed either. The formula for the 88 bits not sent through S-boxes, which are only used as guards, could be solved within seconds, for further five bits we could confirm probing security after 37, 40, 41, 47, and 48 days respectively, but the confirmation for the remaining bits is still open.

Security across several rounds. As described above, our COTG-based design is considered to be probing secure for one round. Although we do not make any security claim beyond one round, our practical evaluations indicate that multiple rounds of our implementation are also secure due to the refreshing performed at two points in the design at the end of every round. First, we add 64 bits of fresh randomness before MixColumns by performing column-wise resharing. Second, AddRoundKey refreshes the complete 128-bit state of the cipher with state-independent key material. The key is completely independent of the state because of the strict separation of guards and fresh randomness for the key schedule and plaintext. However, after two rounds, the key shares and the state cannot be considered completely independent anymore because of the AES key



(a) A sample trace (b) 1st-order t-test

(c) 2nd-order t-test (d) 3rd-order t-test

Figure 5.: Experimental analysis of our masked AES using 100 million traces.

schedule. More concretely, the key bytes are initially completely independent of each other. After executing one round of the key schedule, every key byte will at least depend on one other key byte, the guards used in the S-box, and some randomness. Even though this might lead to a small bias, our practical evaluations using TVLA confirm that this bias is not observable nor exploitable in practice.

6.3. Experimental Verification

In the last section, we discuss the outcome of the formal analysis, which indicates that our design is also second-order secure in the presence of glitches. Since formal verification is limited to less than one round of the design, we show practical evidence for the proposed statements for multiple rounds by porting the design to an FPGA in this section.

Evaluation setup. We perform practical evaluations using a first-, second- and third-order t-test on the NewAE CW305 Artix-7 FPGA evaluation board connected to a PicoScope 6404C at 625 Ms/s sampling rate (1.6 ns sampling interval). The hardware design operates at a clock frequency of 1.5625 MHz, which was chosen as a fraction of the sampling rate. To reduce the noise level, we synchronize the clocks between the FPGA and the oscilloscope and apply a preprocessing step to provide the equal alignment of traces. We implement our complete AES design, including the Trivium RNG as shown in Figure 4, along with some outer control logic used to send and receive data via the USB interface. The implementation receives three shares for the 128-bit plaintext, three shares of the 128-bit key, and a key-IV-pair to initialize the Trivium RNG. The Trivium RNG is initialized once in the beginning and produces 64 bits of fresh randomness per cycle during the encryption. In order to show whether or not a masked implementation exhibits first-order leakage, we follow the standard method and perform Welch's t-test

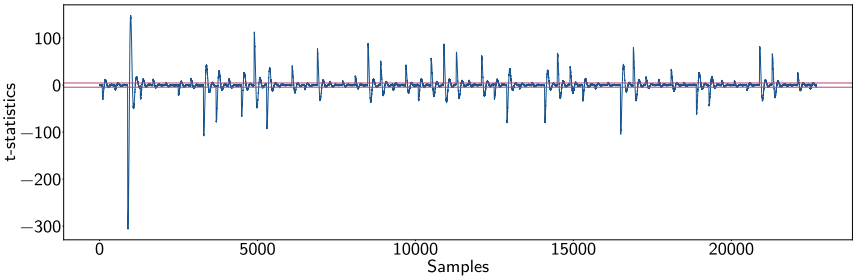


Figure 6.: 1st-order t-test with RNG off and no initial sharing (two shares of plaintext and two shares of key are zeros) using 100 000 traces.

following the guidelines of Goodwill et al. [Goo+11]. The basic idea of the test is to create a random and a fixed set of measurements, one representing the power consumption of the design when processing a random input and one when processing a fixed (constant) input. In order to determine if there are statistically significant differences in the mean power consumption of the two trace sets, one can compute Welch's t-score. The null hypothesis is that both trace sets have equal means, which can be rejected with a confidence greater than 99.999% if the t-score exceeds ±4.5. This implies that the trace sets can be distinguished from each other. A first-order univariate t-test investigates distinguishably on the basis of the mean (first statistical moment) of the trace sets, a second-order univariate t-test uses the variance (second statistical moment) and a third-order univariate t-test uses the third statistical moment.

Discussion. To conduct a first-order, second-order, and third-order t-test, we choose a constant key, for which we generate a new valid sharing for every trace. For the fixed trace set, we set the input plaintext to zero and generate a new valid sharing for every trace of the fixed set. For the random set, we choose all three plaintext shares randomly for every trace. The fixed and random sets are recorded in an interleaved manner, and the RNG is enabled during our measurements. We measured the complete AES encryption, i.e., 10 rounds, as shown in a sample power trace in Figure 5a. The results of the first-order and second-order t-test are given in Figure 5b and Figure 5c. We did not observe evidence for first- or second-order leakage with up to 100 million traces, as the t-score never crosses the ±4.5 threshold. As shown in Figure 5d, we recorded third-order leakage as expected. The t-score exceeded the ±4.5 threshold during the initial AddRoundKey, where the overall noise level is expected to be very low. Since the key schedule starts one cycle before the processing of the plaintext, during the initial AddRoundKey, the processing of the data has not yet started, and the SubWord of the key schedule is only computing the linear mapping. No significant other computations are performed, leading to a low noise level.

To verify the soundness of our setup and to demonstrate that our countermeasure is effective, we show the t-test results of the design without supplying fresh randomness in Figure 6. This means we disable the RNG and the initial sharing of plaintext and key, i.e., two shares of the plaintext and two shares of the key are all zeros. As expected, after 100 000 traces, the design clearly showed first-order leakage.

7. Conclusion

In this work, we presented a second-order masked hardware design of the AES with an improved latency-randomness tradeoff. The resulting round-based (parallel) DOM-masked AES design works with three shares, has a latency of 5 cycles per round, and requires 3 200 random bits per encryption, which can smoothly be delivered by an RNG producing 64 bits of fresh randomness per cycle. The core of our AES design is a masked 5-cycle S-box which requires 78 bits of fresh randomness. We show how randomness can be reused across S-box instances using the COTG technique. We give formal security proofs, conduct an empirical evaluation using TVLA on an FPGA, and compare the implementation cost in terms of area consumption.

Acknowledgments

This work was supported by the FWF SFB project SpyCoDe F8504, and the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We thank our shepherd and anonymous reviewers for their valuable comments in strengthening this work. The authors would like to thank Gaëtan Cassiers for helpful discussions.

References

[ANR19] Victor Arribas, Svetla Nikova, and Vincent Rijmen. "Guards in action: First-order SCA secure implementations of KETJE without additional randomness". In: *Microprocess. Microsystems* 71 (2019).

[Ask+22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. "Guarding the First Order: The Rise of AES Maskings". In: *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*. Ed. by Ileana Buhan and Tobias Schneider. Vol. 13820. Lecture Notes in Computer Science. Springer, 2022, pp. 103–122.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[BDZ20] Tim Beyne, Siemen Dhooghe, and Zhenda Zhang. "Cryptanalysis of Masked Ciphers: A Not So Random Idea". In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 817–850.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 397–426.

[Bey+21] Tim Beyne, Siemen Dhooghe, Adrián Ranea, and Danilo Sijacic. "A Low-Randomness Second-Order Masked AES". In: *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*. Ed. by Riham AlTawy and Andreas Hülsing. Vol. 13203. Lecture Notes in Computer Science. Springer, 2021, pp. 87–110.

[Bie+20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[Bil+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "A More Efficient AES Threshold Implementation". In: *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*. Ed. by David Pointcheval and Damien Vergnaud. Vol. 8469. Lecture Notes in Computer Science. Springer, 2014, pp. 267–284.

[Bil+15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Trade-Offs for Threshold Implementations Illustrated on AES". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 34.7 (2015), pp. 1188–1200.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Can05] David Canright. "A Very Compact S-Box for AES". In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 441–455.

[Can06] Christophe De Cannière. "Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles". In: *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*. Ed. by Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Vol. 4176. Lecture Notes in Computer Science. Springer, 2006, pp. 171–186.

[Cas+23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. "Randomness Generation for Secure Hardware Masking - Unrolled Trivium to the Rescue". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1134.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cnu+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. "Higher-Order Threshold Implementation of the AES S-Box". In: *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*. Ed. by Naofumi Homma and Marcel Medwed. Vol. 9514. Lecture Notes in Computer Science. Springer, 2015, pp. 259–272.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference". In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555. DOI: 10.1109/TIFS.2020.2971153. URL: https://doi.org/10.1109/TIFS.2020.2971153.

[Dae17] Joan Daemen. "Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 137–153. DOI: 10.1007/978-3-319-66787-4_7. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_7.

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon. Submission as a Finalist to the NIST Lightweight CryptoStandardization Process*. https://csrc.nist.gov/Projects/lightweight-cryptography/finalists. Retrieved on July 12th, 2023. 2021.

[DSM22] Siemen Dhooghe, Aein Rezaei Shahmirzadi, and Amir Moradi. "Second-Order Low-Randomness d + 1 Hardware Sharing of the AES". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 815–828.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 1–21.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A testing methodology for side-channel resistance validation". In: *NIST Non-Invasive Attack Testing Workshop*. 2011.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 158–172.

[GPM23] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Formal Verification of Arithmetic Masking in Hardware and Software". In: *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part I*. Ed. by Mehdi Tibouchi and Xiaofeng Wang. Vol. 13905. Lecture Notes in Computer Science. Springer, 2023, pp. 3–32.

[HB21] Vedad Hadzic and Roderick Bloem. "COCOALMA: A Versatile Masking Verifier". In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.34727/2021/isbn.978-3-85448-046-4_9. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4%5C_9.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[JPS18] Bernhard Jungk, Richard Petri, and Marc Stöttinger. "Efficient Side-Channel Protections of ARX Ciphers". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 627–653.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification". In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[low19] lowRISC contributors. *Open Titan*. https://opentitan.org/. Retrieved on March 23th, 2023. 2019. URL: https://opentitan.org/.

[low23] lowRISC contributors. *Open Titan - AES - Theory of Operation*. 2023. URL: https://opentitan.org/book/hw/ip/aes/doc/theory_of_operation.html (visited on 04/12/2023).

[MMM21] Nicolai Müller, Thorben Moos, and Amir Moradi. "Low-Latency Hardware Masking of PRINCE". In: *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*. Ed. by Shivam Bhasin and Fabrizio De Santis. Vol. 12910. Lecture Notes in Computer Science. Springer, 2021, pp. 148–167.

[Moo+19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. "Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 256–292.

[Mor+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES". In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 69–88.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. "Side-Channel Leakage of Masked CMOS Gates". In: *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. Springer, 2005, pp. 351–365.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. "Successfully Attacking Masked AES Hardware Implementations". In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 157–171.

[MRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. "Multiplicative Masking for AES in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 431–468.

[Nag+22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. "Riding the Waves Towards Generic Single-Cycle Masking in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 693–717.

[Nat01] National Institute of Standards and Technology (NIST). *FIPS-197: Advanced Encryption Standard*. 2001. URL: http://www.itl.nist.gov/fipspubs/.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[ODo14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5.

[Osw+05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. "A Side-Channel Analysis Resistant Description of the AES S-Box". In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*. Ed. by Henri Gilbert and Helena Handschuh. Vol. 3557. Lecture Notes in Computer Science. Springer, 2005, pp. 413–423.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427.

[Sas+20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. "Low-Latency Hardware Masking with Application to AES". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 300–326.

[SBM21] Aein Rezaei Shahmirzadi, Dusan Bozilov, and Amir Moradi. "New First-Order Secure AES Performance Records". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 304–327.

[SD17] Niels Samwel and Joan Daemen. "DPA on hardware implementations of Ascon and Keyak". In: *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*. ACM, 2017, pp. 415–424. DOI: 10.1145/3075564.3079067. URL: https://doi.org/10.1145/3075564.3079067.

[Sim+22] Mateus Simoes, Lilian Bossuet, Nicolas Bruneau, Vincent Grosso, Patrick Haddad, and Thomas Sarno. "Self-timed Masking: Implementing Masked S-Boxes Without Registers". In: *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*. Ed. by Ileana Buhan and Tobias Schneider. Vol. 13820. Lecture Notes in Computer Science. Springer, 2022, pp. 146–164.

[SM21] Aein Rezaei Shahmirzadi and Amir Moradi. "Re-Consolidating First-Order Masking Schemes Nullifying Fresh Randomness". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.1 (2021), pp. 305–342.

[SP06] Kai Schramm and Christof Paar. "Higher Order Masking of the AES". In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. Ed. by David Pointcheval. Vol. 3860. Lecture Notes in Computer Science. Springer, 2006, pp. 208–225.

[Sug19] Takeshi Sugawara. "3-Share Threshold Implementation of AES S-box without Fresh Randomness". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 123–145. DOI: 10.13154/tches.v2019.i1.123-145. URL: https://doi.org/10.13154/tches.v2019.i1.123-145.

[Uen+16] Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. "A High Throughput/Gate AES Hardware Architecture by Compressing Encryption and Decryption Datapaths - Toward Efficient CBC-Mode Implementation". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 538–558.

[WM18] Felix Wegener and Amir Moradi. “A First-Order SCA Resistant AES Without Fresh Randomness”. In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 245–262.

[Wol16] Claire Wolf. *Yosys Open Synthesis Suite*. http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021. 2016.

7

Formal Verification of Arithmetic Masking in Hardware and Software

Publication Data. Barbara Gigerl, Robert Primas, and Stefan Mangard. "Formal Verification of Arithmetic Masking in Hardware and Software".
In: *ACNS*. 2023.

Contribution. The author of this thesis contributed to deriving the concepts, performed all the experiments, made the implementations and wrote large parts of the text.

Formal Verification of Arithmetic Masking in Hardware and Software

Barbara Gigerl[1], Robert Primas[1], Stefan Mangard[1,2]

[1] Graz University of Technology [2] Lamarr Security Research

Abstract Masking is a popular countermeasure to protect cryptographic implementations against physical attacks like differential power analysis. So far, research focused on Boolean masking for symmetric algorithms like AES and Keccak. With the advent of post-quantum cryptography (PQC), arithmetic masking has received increasing attention because many PQC algorithms require a combination of arithmetic and Boolean masking and respective conversion algorithms (A2B/B2A), which represent an interesting but very challenging research topic. While there already exist formal verification concepts for Boolean masked implementations, the same cannot be said about arithmetic masking and accompanying mask conversion algorithms.

In this work, we demonstrate the first formal verification approach for (any-order) Boolean and arithmetic masking which can be applied to both hardware and software, while considering side-effects such as glitches and transitions. First, we show how a formal verification approach for Boolean masking can be used in the context of arithmetic masking such that we can verify A2B/B2A conversions for arbitrary masking orders. We investigate various conversion algorithms in hardware and software, and point out several new findings such as glitch-based issues for straightforward implementations of Coron et al.-A2B in hardware, transition-based leakage in Goubin-A2B in software, and more general implementation pitfalls when utilizing common optimization techniques in PQC. We provide the first formal analysis of table-based A2Bs from a probing security perspective and point out that they might not be easy to implement securely on processors that use of memory buffers or caches.

1. Introduction

Passive side-channel attacks, including power or electromagnetic analysis, are among the most relevant attack vectors against cryptographic devices like smart cards, that are physically accessible by an attacker [KJJ99; QS01]. A commonly used approach to protect against these attacks is to implement algorithmic countermeasures, for example masking [Cnu+16; GIB18; GMK16; ISW03; Rep+15]. Masking schemes split input and intermediate values of cryptographic computations into $d + 1$ random shares such that observations of up to d shares do not reveal any information about the native (unmasked) value. Boolean masking,

where native values correspond to the XOR-sum over its shares, have received much attention since such schemes are applicable to almost all symmetric cryptographic algorithms. On the other hand, arithmetic masking schemes have also gained increased importance, especially with the advent of PQC, for which they generally represent a better fit. In arithmetic masking, native values correspond to the arithmetic addition over their shares which allows to express operations like addition and subtraction much more efficiently than with Boolean masking. Since PQC algorithms often use symmetric building blocks, e.g. to achieve CCA2-security or for sampling random numbers, arithmetic masking often has to be combined with Boolean masking [Bos+21; FO99; Sch+19], which requires efficient and secure A2B/B2A conversion techniques. Many works have shown that hardware side-effects like glitches or transitions can violate the security of masking schemes in practice. Hence, the design of masked cryptography requires a detailed understanding of the targeted hardware platform, and is therefore a notoriously error-prone and time consuming task. Consequently, there is strong need for verification tooling that supports this effort to the highest possible extend.

While there already exists a vast amount of literature on the verification of Boolean masking, including formal verification approaches like Rebecca[Blo+18], maskVerif [Bar+19], Coco (Alma) [Gig+21; HB21], SILVER [KSM20], or scVerif [Bar+21], the same cannot be said about arithmetic masking.

Limitations of Existing Approaches The first works on formal verification of arithmetic masking schemes were published with QMVerif by Gao et al. [Gao+19a] and `LeakageVerif` by Meunier et al. [MPH21]. These works already form a good foundation, but are limited in several ways.

QMVerif was published in 2019 by Gao et al. [Gao20] for the verification of first-order Boolean and arithmetically masked software. QMVerif uses *type inference* to determine the distribution of every variable in the masked software, which is either uniform, independent of private inputs or dependent on private inputs. Due to the lack of completeness guarantees of type inference, deducing the distribution might not always be possible and might lead to false positives [MPH21]. In that case, QMVerif uses *model-counting* to compute the exact distributions using a SAT solver, which is complete, but does not scale and consequently often requires significant computational resources such as GPU acceleration [Gao+20]. This scalability issue leads to the conclusion that model-counting-based masking verification is generally infeasible in the context of arithmetic masking. Besides that, QMVerif does not support masked hardware and heavily restricts supported masked software, e.g. by requiring a specific high-level syntax, not allowing branches, loops or functions and limiting variables to 8 bit. The leakage model of QMVerif hence also does not consider hardware side-effects like glitches and transitions, and it is unclear wether QMVerif can be applied to A2B/B2A conversions without a power-of-two-modulus. The same

authors later propose HOME for higher orders following the same approach, but do not evaluate it for higher-order arithmetic masking. Since the tool is not (yet) open-source, it is not possible to investigate its further functionality.

Meunier et al. [MPH21] propose `LeakageVerif`, a Python verification library based on *substitution* [Bar+15], which tries to show that an expression is leakage-free if it can be divided into sub-expresions, which can iteratively be substituted by fresh random variables. The evaluation shows that `LeakageVerif` is more efficient than QMVerif, but it is not complete and fails to verify common A2B/B2A conversions such as Goubin-A2B [Gou01] and Coron et al.-B2A. `LeakageVerif` works for first-order implementations only, does not consider glitches, table lookups, or moduli which are not a power of two.

In general, both QMVerif and `LeakageVerif` are *sound* (leakages are never missed), but can sometimes only achieve *completeness* (leaks are only reported if they really exist) if they fall back to expensive and inefficient model-counting.

Other existing verification tools focus exclusively on Boolean masking and often perform exact model-counting, which are therefore unlikely to be applicable to arithmetic masking. For example, maskVerif has been shown infeasible in this context by several works [Gao+19b; Gao+20; MPH21]. In 2021, Bos et al. present scVerif [Bar+21], which was later modified for the verification of a first-order arithmetically masked software implementation of Kyber [Bos+21]. However, scVerif was not evaluated for other arithmetically masked programs, so no general statement about its efficiency or accuracy can be made. It does consider hardware side-effects but only if they have been identified in prior empirical experiments, which means the method is not sound and binds the evaluation stronger to the microarchitecture, while leaving no potential for masked hardware. We expect SILVER [KSM20] to also not be able to deal with the complexity of arithmetic expressions since it exclusively tracks exact distributions with the help of binary decision diagrams.

In 2018, Bloem et al. suggest to approximate Fourier coefficients of Boolean functions [Blo+18] as a way to perform cheaper model counting that achieves soundness but not completeness. The resulting approach was evaluated for Boolean masked hardware (Rebecca), and later for software on CPU netlists (Coco) [Gig+21; HB21], and has shown to be efficient with a relatively low rate of false positives. However, it was not evaluated for arithmetic masking in terms of efficiency, accuracy, and general applicability for PQC-relevant use cases.

Our Contribution We improve this situation by demonstrating that the security of arithmetically masked software/hardware can be efficiently verified using verification approaches tailored to Boolean masking. More concretely, we provide the following contributions:

- We show how verification methods based on approximated Fourier coefficients of Boolean functions (as used by Rebecca/Coco) can be efficiently applied in the context of arithmetic masking. The resulting verification

approach can successfully be applied to both masked hardware and software written in Assembly language. Its soundness is sufficient for many PQC/ARX applications. This approach is also the first to consider physical defaults (glitches, transitions) and the first to be evaluated for higher orders in the context of arithmetic masking (Section 3).

- In case of hardware implementations, we analyze different versions of the Coron et al.-A2B/B2A [CGV14] conversion algorithms and identify potential weaknesses caused by glitches. We then present a proof-of-concept implementation that is secured against glitches and can be fully verified using our approach (Section 4).

- In the context of software implementations, we analyze various popular A2B/B2A conversion algorithms using power of two or prime moduli and provide new insights on implementation aspects that can reduce their protection order. More concretely, we report new findings of transition leakages in Goubin-A2B [Gou01] and point out more general pitfalls when using lazy-reduction techniques in the context of masking. Additionally, we are the first to investigate architecture side-effects of table-based A2Bs and discuss why they might not be easy to implement securely on processors that make use of memory buffers or caches. Last but not least, we also show applicability of this approach in the context of symmetric cryptographic schemes by verifying the security of masked software implementations of one round of Speck and the ARX-box Alzette(Section 5).

- We plan to publish the software and hardware implementations on Github[1].

2. Background

In this section, we cover necessary background on masking, and A2B/B2A conversion techniques. Since our approach is based on Rebecca/Coco, we briefly describe the verification concept and the applied adversary model.

2.1. Masking Schemes and Applications

Masking is a prominent algorithmic countermeasure against Differential Power Analysis [KJJ99] that splits intermediate values of a computation into $d+1$ uniformly random shares [Cnu+16; GIB18; GMK16; ISW03], such that an attacker who observes up to d shares cannot deduce information about native (unshared) intermediate values. Boolean masking is commonly used for symmetric cryptography, and uses the exclusive or ($\oplus$) operation to split a value b into $d+1$ uniformly random shares $b_0 \dots b_d$ such that $b = \bigoplus_i b_i = b_0 \oplus \cdots \oplus b_d$. In arithmetic masking schemes, the relation between shares of a value a is the

[1] https://github.com/barbara-gigerl/arithmetic-masking-hw-sw

modular addition, yielding $a = \sum_i a_i = a_0 + \cdots + a_d \mod q$. In both cases, masking linear functions is trivial since they can simply be computed for each share individually. Masking non-linear functions is more challenging since these functions operate on all shares of a native value and thus usually require additional fresh randomness to avoid unintended direct combination of shares. The concrete technique (Boolean or arithmetic) determines which operations are (non-)linear.

PQC algorithms often perform operations like matric/polynomial multiplication, which can be efficiently masked in the arithmetic domain when broken down into coefficient-wise modular addition/multiplications using e.g. the number theoretic transform (NTT). In practice, arithmetic masking often has to be combined with Boolean masking since building blocks including Gaussian samplers and lattice decoding, or constructions like the Fujisaki-Okamoto transform for achieving CCA2-security are more efficiently masked in the Boolean domain. Therefore, many masked implementations use dedicated conversion algorithms to transform shares from the arithmetic to the Boolean domain (A2B) and vice versa (B2A). Besides PQC, arithmetic masking is also applied to ARX-based implementations like SHA-256 or ChaCha, but comes with a significantly higher runtime overhead compared to Boolean masked variants of non-ARX symmetric algorithms.

2.2. Mask Conversion Techniques

Many cryptographic schemes require to switch between the Boolean and the arithmetic domain when respective masking techniques are applied. The performance of the protected scheme is mainly determined by the A2B and B2A conversions used, which is why there has been a lot of research in this direction [CGV14; Cor+12; Cor+15; Cor+22; Cor17; Sch+19; SMG15]. Existing conversion algorithms either follow an *algebraic* or a *table-based* approach. An algebraic conversion algorithm performs the whole conversion at once, while table-based approaches first pre-compute a table which is later used during the actual conversion. B2A conversions can be done very efficiently following the algebraic approach, while A2B is less efficient, and therefore often apply a table-based approach.

In 2001, the first algebraic conversion algorithms were proposed by Goubin [Gou01], and by Coron et al. [CGV14] for higher orders. They propose the SecAdd algorithm, which allows to securely add Boolean shares at any order using a power-of-two modulus. Many follow-up works use Coron et al.-A2B/B2A as a basis, and suggest several performance improvements [BCZ18; Cor+15; Cor17; HT19]. Since PQC applications often require a prime modulus, Barthe et al. [Bar+18], and later Schneider et al. [Sch+19] suggest how to adapt Coron et al.-B2A to work with prime moduli.

Table-based A2B conversion algorithms use pre-computed tables to reduce the computation effort during the actual conversion. In general, A2B conversions transform the shares together with the carry which is produced in an arithmetic

addition. The pre-computed tables are used to handle the conversion of the carry, and prevent unintended unmasking of native values. The first table-based A2Bs were suggested by Coron-Tchulkine [CT03] and Neiße-Pulkus [NP04]. They were however shown to be incorrect and insecure by Debraize [Deb12], who suggests several corrected and optimized versions of their algorithms. Recently, Beirendonck et al. [BDV21] show that Debraize-A2B does also not fulfill its security claims, and propose two further table-based A2Bs.

A2B/B2A conversions are applied to masked implementations of various PQC and ARX schemes against side-channel attacks. For example, the SecAdd algorithm by Coron et al. [CGV14] has been used as a cryptographic primitive in several software [AFM17; Bar+18; Bos+21; Cor+15; GR19; Sch+19] and hardware implementations [Che+15; Fri+22]. Debraize-A2B has also been applied recently in works on masking PQC [Bos+21; Ode+18].

2.3. Masking Verification with Rebecca/Coco

Rebecca [Blo+18] is a tool to formally verify Boolean-masked hardware implementations defined by gate-level netlists. In order to verify a circuit, a label is assigned to each circuit input. The label is either a share, fresh randomness or unimportant. During the verification process, these labels are propagated through the circuit and each gate is assigned a correlation set according to the propagation rules. In general, a correlation set contains information about the statistical dependence of the respective gate on the circuit inputs. Tools like Silver [KSM20] compute these dependencies accurately, while Rebecca approximates statistical dependence with non-zero Fourier coefficients [Blo+18]. In general, the Fourier or Walsh expansion of a boolean function refers to the representation of the function as a multilinear polynomial [ODo14]. A term in the polynomial, which is either a label or a combination of labels, with a non-zero Fourier coefficient indicates statistical dependence on the respective circuit input. The approximation is performed by not tracking the exact Fourier coefficient, but only whether a term has a non-zero coefficient or not. A correlation set contains all terms with non-zero coefficients.

Later, an optimized variant of this approach was implemented in Coco, a tool for the formal verification of (any-order) Boolean masked software implementations on concrete CPUs [Gig+21; GPM21]. The main purpose of Coco is to analyze the potential implications of hardware side-effects like glitches within a CPU on masked software implementations. Coco can additionally incorporate control flow logic, which is required for the verification of software and iterative hardware circuits. Before the verification, the CPU netlist is simulated together with a masked assembly implementation, in order to obtain a trace of the (constant) data-independent control signals like memory/register access patterns and branches. Next, similar to Rebecca, initial labels are assigned to registers and memory locations, which are further propagated through the netlist for multiple cycles to construct correlation sets, while considering software-specific control signals. The verification fails if there exists a gate in the netlist which directly

correlates with a native value. In that case, Rebecca reports the leaking gate, while Coco additionally reports the exact clock cycle.

2.4. Adversary Model

The robust probing model for hardware [Fau+18; ISW03] allows an attacker to observe the values of up to d wires in a masked circuit using (g, t, c)-extended probing needles to optionally include glitches ($g = 1$), transitions ($t = 1$) or coupling ($c = 1$). The circuit is dth-order secure if the adversary is not able to learn anything about the native value by combining these observations. Accordingly, the standard probing model for software allows the adversary to probe intermediate program variables.

In this work we use the so-called *time-constrained probing model* [Gig+21], which is currently adopted by Coco for masked software implementations. The main difference to the robust probing model is the time restriction of each probe to one clock cycle, which is necessary to correctly model the execution of masked software on netlist level. More concretely, in the time-constrained probing model the attacker uses $(g, t, 0)$-extended probes to observe the value of any specific gate/wire in the CPU netlist for the duration of one clock cycle. The gate/wire and cycle can be chosen independently for each probe.

The time-constrained probing model can be applied to masked hardware circuits and allows to handle *iterative* circuits directly without the need to perform *unrolling* thanks to its time-awareness. In Appendix A we give an example of an iterative circuit and its unrolled version based on the suggestion of [Bha+10]. Verification approaches adopting the classic/robust probing model usually unroll the processed iterative circuit, which works well for simple circuits, but is more difficult for circuits with more complex control logic, such as state machines. Iterative circuits can be seen as a reduced version of a CPU, and therefore allows the direct application of the time-constrained probing model.

The original version of Coco provides two different verification modes in the time-constrained probing model. *Stable* verification focuses on pure algorithmic security. *Transient* verification uses $(g, t, 0)$-extended probes, and therefore considers algorithmic security and wire/register transitions and glitches within the hardware. For the purpose of this work, we add a third mode, the *Transitions* verification mode, working with $(0, t, 0)$-extended probes, which is convenient since it reports stable and transition leaks, but without the runtime overhead of the transient mode.

3. Verification of Arithmetic Masking in the Boolean Domain

In this section, we explain how one can perform verification of arithmetic masking using a method based on approximating Fourier coefficients of Boolean functions

that was previously used by the tools Rebecca/Coco in the context of Boolean masking. In Section 3.1 we recall how arithmetic expressions can directly be broken down into equivalent Boolean expressions on bit-granularity. In Section 3.2 we discuss optimization strategies that can be used to reduce the complexity of the derived Boolean expressions for the initial labeling and to more efficiently propagate expressions through dedicated arithmetic addition circuits. We also comment on the soundness and completeness of our approach. Finally, we give a small self-contained example in Section 3.3.

Notation

We denote with $a^{(i)}$ the i-th bit of variable a, with a^0 being the least significant bit (LSB). The j-th share of a native variable x is identified as x_j. Similar to Bloem et al. [Blo+18], we denote the correlation set of a gate/wire w by $\mathcal{C}(w) = \{...\}$. As introduced in [Gig+21], the $\otimes$-operator computes the element-wise multiplication of two correlation sets. We use small letters for symbolic expressions, while capital letters are used for wires in a circuit.

3.1. Modeling Arithmetic Expressions using Boolean Logic

A netlist represents a circuit design after logic synthesis that models gates as Boolean functions mapping 1-bit inputs to a 1-bit output, and indicates their interconnection. We aim at performing netlist-level verification of a circuit on bit granularity. In the end, a bitwise view on all terms computed by the circuit must still valid in the context of masking. This implies that the dependencies between the shares must be described using Boolean equations on bit granularity. Such a mapping can be obtained based on the definition of the Ripple-carry adder, which represents a cascade of 1-bit full adders, where each carry bit *ripples* to the next full adder. Each full adder takes two 1-bit summands and a 1-bit carry-in, and computes the arithmetic sum and respective carry-out [Man82].

Consider a sum s, which is computed from the summands u and v such that $s = u + v$. If u and v are n-bit values, s is represented by $n + 1$ bits, and hence, $n + 1$ full adders are needed to compute s. Each full adder takes two summand bits $u^{(i)}$ and $v^{(i)}$ together with the carry-in $c^{(i)}$, and computes $s^{(i)}$ as:

$$s^{(i)} = u^{(i)} \oplus v^{(i)} \oplus c^{(i)} \text{ with } c^{(0)}=0 \tag{1}$$

The carry-out bit $c^{(i+1)}$ is then computed based on the carry-in $c^{(i)}$ by the following recursive formula:

$$c^{(i+1)} = (u^{(i)} \oplus v^{(i)}) \wedge c^{(i)} \vee (u^{(i)} \wedge v^{(i)}) \tag{2}$$

Equation 1 already gives a valid first-order Boolean sharing for s using the two shares $x_1 = u$ and $x_2 = v \oplus c$.

If a sum t is split into three summands u, v and w such that $t = u + v + w$, basically the same equations apply, and t can be computed in two steps. In the first step, the partial sum $s = u + v$ is computed, which yields the carry c. In the second step, t is computed by adding the partial sum to the remaining summand: $t = s + w$, which produces the carry e:

$$t^{(i)} = s^{(i)} \oplus w^{(i)} \oplus e^{(i)} \tag{3}$$

$$= u^{(i)} \oplus v^{(i)} \oplus c^{(i)} \oplus w^{(i)} \oplus e^{(i)} \tag{4}$$

Equation 4 gives a valid second-order Boolean sharing for t using three shares $x_1 = u, x_2 = v$ and $x_3 = w \oplus c \oplus e$. Formulas for more than three summands can be derived in a similar way, each resulting in a valid higher-order sharing. When working with $d + 1$ shares, the first d Boolean shares would always be equal to the first d arithmetic shares, while the last Boolean share needs to additionally include the carry.

3.2. Tailoring the Verification Approach

Arithmetically masked circuits process arithmetic input shares, while Rebecca/Coco expects Boolean input shares. The derived Boolean equations for arithmetic expressions in Section 3.1 can now be used to translate arithmetic shares to the Boolean domain, such that Rebecca/Coco could work with it. In the following, we describe how one can obtain such a translation in a correct and efficient way, how the resulting expressions can be propagated efficiently, and comment on soundness, completeness and scalability of the resulting approach.

Initial Labeling

Tools for the formal verification of masking require a set of initial labels that specify the location/dependency of shares on circuit inputs, registers or memory cells that are then further tracked throughout a circuit. In the case of (first-order) Boolean masking, each bit of a native value $a^{(i)}$ is initially masked with a random mask $r^{(i)}$. Therefore, the native value $a^{(i)}$ can simply be expressed as the XOR between the two shares $a^{(i)} \oplus r^{(i)}$ and $r^{(i)}$. As shown in Figure 1, the labels assigned prior to the verification would then be $b_0^{(i)} = a^{(i)} \oplus r^{(i)}$ and $b_1^{(i)} = r^{(i)}$.

In the case of (first-order) arithmetic masking, each bit of a native value $a^{(i)}$ is initially masked with a random mask $r^{(i)}$ using modular additions. According to Equation 1, the native value $a^{(i)}$ can be expressed as the XOR between the two shares $a^{(i)} \oplus r^{(i)} \oplus c^{(i)}$ and $r^{(i)}$. In contrast to Boolean masking, we also need to include the carry of the addition $c^{(i)}$, which depends on lower bits of $a^{(i)}$ and $r^{(i)}$. The first option to obtain a valid labeling for arithmetic shares is thus to resolve $c^{(i)}$ recursively according to Equation 2. The initial labels would

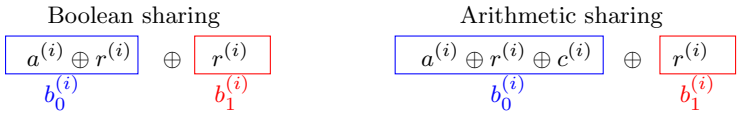


Figure 1.: Initial labeling for Boolean and arithmetic masking as given to the verifier

then be given by $b_0^{(i)} = (a^{(i)} \oplus r^{(i)}) \oplus c^{(i)}$, and $b_1^{(i)} = r^{(i)}$. Here, the carry $c^{(i)}$ is computed recursively for each bit position, which adds already quite complex terms to the correlation set at the beginning of the verification, especially for the more significant bits of the arithmetic shares since the depend in a non-linear way on all lower bits.

It is however also possible to use a different initial labeling that incorporates additional information that is available at the beginning of the verification and significantly simplifies the resulting Boolean expressions. More concretely, with each $c^{(i)}$ being a non-linear combination of all lower bits (including their masks), this expression alone must never be observable by an attacker. Put differently, each bit of a fresh arithmetic share is only independent of any native values because the term $r^{(i)}$ is added in a linear way and does not occur in any of the lower bits (and thus also not $c^{(i)}$). It is hence sufficient to verify if the linear term $r^{(i)}$ in a certain bit of one arithmetic share ever gets in contact with the same $r^{(i)}$ in the corresponding bit of the other share, similarly as in the case of Boolean masking (c.f. Figure 1). This simplification leads to simpler expressions for the initial labels and thus improves verification runtime. Note that this simplification is only used for deriving initial labels but not during mask refresh operations throughout the masked computation where our assumptions on unique usage of fresh randomness does not necessarily hold anymore. This simplification also applies to initial labels of higher order arithmetic masking in a similar manner.

Fourier Expansion of Arithmetic Addition

One particularly challenging aspect of verifying arithmetic masking is scalability due to complex dependencies between shares on bit-level, introduced by the carry when an arithmetic addition is computed. In hardware, arithmetic additions are often performed by dedicated sub-circuits. For example, CPUs usually have such an adder circuit in their ALU (Arithmetic Logic Unit). In Equation 5 we propose the Fourier expansion W of arithmetic additions, which allows to directly obtain correlation sets for the result of an adder circuit, instead of computing an individual correlation set for every gate within the adder, and thus speeds up the verification runtime. The expansion of the sum is based on the Fourier expansion of the carry given in Equation 6.

$$W(s^{(j)}) = \frac{1}{2}u^{(j)} \cdot v^{(j)} \cdot c^{(j)} + \frac{1}{2}u^{(j)} + \frac{1}{2}v^{(j)} - \frac{1}{2}W(c^{(j)}) \quad (5)$$

$$W(c^{(j)}) = \frac{1}{2}W(c^{(j-1)}) + \frac{1}{2}v^{(j)} + \frac{1}{2}u^{(j)} - \frac{1}{2}u^{(j)} \cdot v^{(j)} \cdot W(c^{(j-1)}) \quad (6)$$

In Section 5.1 we give more details about how this can be used to increase the performance of software verification. More details on how we derived both expansions are given in Appendix B.

Soundness and Completeness

While masking verification based on approximated Fourier coefficients of Boolean functions is sound (leakages are never missed), it is not complete (leaks might be reported although the implementation is secure). Throughout a masked computation it might happen that certain terms in the exact Fourier representation cancel out or evaluate to constants. Our verification approach might miss such situations since it only keeps track of whether a term occurs in a correlation set or not (for performance reasons), which ultimately results in an overapproximation of the exact Fourier representation. If a situation occurs in which e.g. multiple shares with a correlation coefficient of zero are combined, the verifier would report a leak that does not exist in practice (which implies non-completeness). Soundness is however guaranteed by the fact that the verifier always keeps track of an *over*approximation of all the terms that a register/wire could depend on, hence, a real leak can never be missed.

In case of sound but not complete masking verification approaches, the amount of false positive leakage reports in realistic scenarios plays an important for practicality. Simply speaking, the longer a computation becomes, the more likely a false positive occurs. Note however that after every mask refresh operation, the newly introduced randomness essentially eliminates possible future false-positive leaks caused by over-approximation that has happened thus far. In other words, as long as mask refreshing occurs somewhat frequently (which is generally the case) the occurrence of false positive leak reports will generally be quite low. Later, in Section 4 and Section 5, we show that the soundness of our approach is in fact sufficient to perform meaningful verification of masked SW/HW implementations in many typical PQC/ARX applications.

During our analysis in this work, we only really observe a single false positive when verifying Goubin-A2B [Gou01] in software, as discussed in more detail in Section 5.2.

3.3. Example

Finally, we give an example about how correlation sets are constructed using our verification approach. Assume an example circuit which takes two 2-bit arithmetic

shares $a + r$ (input signal A_0) and r (input signal A_1), and two bits of fresh randomness s (input signal S). The ultimate goal is to compute $(A_0 + S) + A_1$ by using two *Full Adder*s. In order to verify the first-order security of this circuit, one first assigns the respective labels to the inputs which result in the following correlation sets:

$$\mathcal{C}(A_0^{(0)}) = \{\{b_0^{(0)}\}\}, \quad \mathcal{C}(A_1^{(0)}) = \{\{b_1^{(0)}\}\}, \quad \mathcal{C}(S^{(0)}) = \{\{s^{(0)}\}\},$$
$$\mathcal{C}(A_0^{(1)}) = \{\{b_0^{(1)}\}\} \quad \mathcal{C}(A_1^{(1)}) = \{\{b_1^{(1)}\}\} \quad \mathcal{C}(S^{(1)}) = \{\{s^{(1)}\}\}$$

The input bits are propagated to the first adder, which computes $(A_0 + S)$. We obtain the following correlation sets at the output signals of the first adder:

$$\begin{aligned}
\mathcal{C}(\text{Adder1}_{sum}^{(0)}) &= \mathcal{C}(A_0^{(0)}) \otimes \mathcal{C}(S^{(0)}) = \{\{b_0^{(0)}, s^{(0)}\}\} \\
\mathcal{C}(\text{Adder1}_{sum}^{(1)}) &= \mathcal{C}(A_0^{(1)}) \otimes \mathcal{C}(S^{(1)}) \otimes \mathcal{C}(\text{Adder1}_{carry}^{(1)}) \\
&= \{\{b_0^{(1)}, s^{(1)}\}\} \otimes \{\{1\}, \{b_0^{(0)}\}, \{s^{(0)}\}, \{b_0^{(0)}, s^{(0)}\}\} \\
&= \{\{b_0^{(1)}, s^{(1)}\}, \{b_0^{(1)}, s^{(1)}, b_0^{(0)}\}, \{b_0^{(1)}, s^{(1)}, s^{(0)}\}, \\
&\quad \{b_0^{(1)}, s^{(1)}, b_0^{(0)}, s^{(0)}\}\} \\
\mathcal{C}(\text{Adder1}_{sum}^{(2)}) &= \mathcal{C}(\text{Adder1}_{carry}^{(2)})
\end{aligned}$$

Note that the second bit of the adder has to be labeled with the (recursively resolved) carry of the addition. These correlation sets are then propagated to the second adder:

$$\begin{aligned}
\mathcal{C}(\text{Adder2}_{sum}^{(0)}) &= \mathcal{C}(A_1^{(0)}) \otimes \mathcal{C}(\text{Adder1}_{sum}^{(0)}) = \{\{b_1^{(0)}, b_0^{(0)}, s^{(0)}\}\} \\
\mathcal{C}(\text{Adder2}_{sum}^{(1)}) &= \mathcal{C}(A_1^{(0)}) \otimes \mathcal{C}(\text{Adder1}_{sum}^{(1)}) \otimes \mathcal{C}(\text{Adder2}_{carry}^{(1)}) \\
&= \{\{b_1^{(1)}, b_0^{(1)}, s^{(1)}\}, \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, b_0^{(0)}\}, \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, s^{(0)}\}, \\
&\quad \{b_1^{(1)}, b_0^{(1)}, s^{(1)}, b_0^{(0)}, s^{(0)}\}\} \\
\mathcal{C}(\text{Adder2}_{sum}^{(2)}) &= \mathcal{C}(\text{Adder1}_{sum}^{(2)}) \otimes \mathcal{C}(\text{Adder2}_{carry}^{(2)}) \\
\mathcal{C}(\text{Adder2}_{sum}^{(3)}) &= \mathcal{C}(\text{Adder2}_{carry}^{(2)})
\end{aligned}$$

Obviously, $(A_0 + S) + A_1$ is a valid operation in the context of arithmetic masking and this is also visible on bit-level. The computation of the carry bits of the second adder combines shares in a non-linear way, which typically leads to a leak. However, the addition is still secure in the end since $(A_0 + S)$ adds randomness to each share bit linearly. When performing an addition of two operands we always conservatively label one bit more than the size of the largest operand to correctly capture bit width of the result independently on the concrete input values. Note that by performing modular reduction one can clear the carry residing in the most significant bit (MSB). This type of computation occurs very

frequently in the beginning of A2B algorithms when two arithmetic shares should be added since the addition of fresh randomness is equivalent to a mask refreshing operation.

4. Application to Masked Hardware Implementations

In this section we apply our verification approach to hardware implementations of Coron et al.-A2B. While it has already been shown in the past that this algorithm is secure in the stable setting, which is also confirmed by our verifier, we want to put our focus mainly on settings where we also consider transition and glitch effects. We show, both via a formal analysis, and in empirical evaluations, that hardware side-effects can reduce the protection order of the implementation. While the straight-forward approach of adding additional register stages whenever needed can eliminate this problem, we also want to point out that this comes with a noticeable increase of latency.

Coron et al.-A2B/B2A

In 2014, Coron et al. [CGV14] have proposed the first higher-order mask conversion algorithm, which we refer to as Coron et al.-A2B/B2A in the following. This algorithm is based on the SecAdd function and allows to perform arithmetic additions in a Ripple-carry fashion on Boolean shares. More specifically, the algorithm converts the arithmetic shares a_0 and a_1 which correspond to the native value a into the Boolean shares b_0 and b_1. The conversion starts with the *initial remasking*, where the arithmetic input shares are refreshed by adding fresh randomness, followed by the *carry computation*. A single native carry bit is computed based on Equation 2, which can be rewritten as:

$$c^{(i+1)} = u^{(i)} \wedge v^{(i)} \oplus u^{(i)} \wedge c^{(i)} \oplus v^{(i)} \wedge c^{(i)} \text{ with } c^{(0)} = 0 \tag{7}$$

However, the algorithm operates on shared carries c_0 and c_1 instead on the native c, which are computed bit by bit using secure masked AND gadgets (SecAnd). In Appendix C we sketch the structure of Coron et al.-A2B when implemented in hardware. The corresponding B2A conversion chooses the first arithmetic share a_0 randomly, and computes $a_1 = (b_0 \oplus b_1) - a_0$ using SecAdd. The algorithm is very efficient for hardware implementations [Fri+22], since SecAdd can be used for both A2B and B2A, and both can also be applied to higher orders. In Section 5 we formally evaluate both Coron et al.-B2A, and a second-order masked software implementation of Coron et al.-A2B.

4.1. Formal Analysis

We implement Coron et al.-A2B with 16-bit shares in hardware. We store all inputs in registers, and implement the remaining parts as a pure combinatorial circuit, which takes a single cycle to finish and therefore does not require a state machine. The input shares as well as the necessary random values are stored in registers. The verifier confirms algorithmic security for this single-cycle implementation, while in the transient case under the consideration of glitches, first-order side-channel protection is not given. More concretely, glitches in the initial remasking phase and the SecAnd modules, which are part of the bigger SecAdd, may lead to a temporary combination of shares due to delayed addition of randomness.

Initial Remasking

Two XOR gates at the circuit's inputs are used to perform the refreshing during the initial remasking phase, which each combines an arithmetic share with a random value. In the worst case, a glitch at the output of the XOR gate propagates the pure values of a_0 and a_1, for example, when the wire delay of the random values is bigger than the wire delay of the arithmetic shares. The input of SecAdd will then be the arithmetic shares without refreshing, and the circuit computes $a_0 + a_1 = a$ for a short time frame in the beginning of the clock cycle, until all wires stabilize and the randomness *arrives* at the gates. As a solution, we add a single additional register stage to store the result of these XOR computations. This ensures that the SecAdd module's input comes out of a register instead of combinatorial logic, and will therefore not glitch.

SecAnd

Coron et al. suggest to use the masked AND gadget proposed by Ishai et al. [ISW03], called ISW-AND, in the SecAnd-blocks of the conversion. Formal verification however reports a leak due to glitches in the SecAnd module because the ISW-AND is not glitch-resistant, and also does not fulfill the required composability properties. As a solution, we suggest to insert two register stages to the SecAnd component. Works like [Cor+13; Fau+18; Moo+19; MPG05] confirm our observation that these two register stages are indeed needed in this case. Combined with the register stage inserted for the initial remasking, this results in a high latency overhead, i.e., for n-bit input shares, the implementation now requires $34 = 2 + 2 \times n$ cycles to complete, and also utilizes a state machine in order to control the execution.

Using this case study, we evaluate our verification approach for masked hardware circuits in Table 1 by comparing the broken single-cycle implementation to the one

Table 1.: Verification of Coron et al.-A2B (broken and fixed) in hardware

Algorithm	Input shares	Runtime (cycles)	Verification result/runtime					
			Stable		Transitions		Transient	
Coron et al. [CGV14]	16 bit	1	✔	11 s	✔	10 s	✖	1 s
Coron et al. [CGV14]	16 bit	34	✔	56 s	✔	2 min	✔	3 min

which adapts our fixes. All experiments are run using a 64-bit Linux Operating System on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. The security on algorithmic level of both implementations can be shown in 11 seconds and respectively 56 seconds in the stable case. We need around a second to find the issues in the transient case, and about three minutes to prove that our fixes indeed provide first-order protection. Our implementation serves the purpose of a proof-of-concept and allows further extensive optimizations. However, we consider the discussion of these optimizations along with the evaluation of area and performance overhead out of scope for this paper.

4.2. Empirical analysis

In the last section we discussed the outcome of the formal analysis which indicates that glitches in the design are problematic in the context of masking. As a second step, we show practical evidence for the proposed statements.

Evaluation Setup

We practically evaluate Coron et al.-A2B using a first-order t-test on the NewAE CW305 Artix-7 FPGA evaluation board connected to a PicoScope 6404C at 312.5 Ms/s sampling rate. The hardware design operates at a clock frequency of 1 MHz. In order to detect potential first-order leakage, we perform Welch's t-test following the guidelines of Goodwill et al. [Goo+11], which is a standard method to measure information leakage of masked implementations. The basic idea is to create two sets of measurements, one representing the power consumption of the design with random inputs (random set), and one with constant inputs (fixed set). For the fixed set, we assume the native value a is 0 and generate the respective shares a_0 and a_1 such that $a_0 + a_1 = a$. For the random set, we generate a_0 and a_1 completely at random. We use fresh random values for the random inputs in both cases. From these trace sets, one can compute Welch's t-score to measure the significance of the difference of means of the two distributions. The null-hypothesis is that both trace sets have equal means, which is rejected with a confidence greater than 99.999% if the absolute t-score does not exceed 4.5, and implies that the trace sets cannot be distinguished from each other.

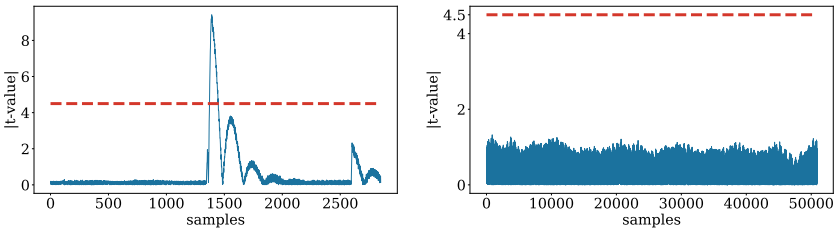


Figure 2.: T-test scores of the original (left) and the secured (right) implementation of Coron et al.-A2B using 400 000 power traces

Discussion

Figure 2 shows our leakage assessment using 400 000 traces. The results for the original, unprotected single-cycle implementation are presented on the left. The t-test score shows significant peaks over the 4.5 border, indicating first-order leakage. On the right side, the leakage evaluation of our 34-cycle fixed implementation is shown, in which the t-score does not cross the significance boarder. Thus, these measurements confirm the security claim made by the formal tool. In Appendix D we show the functionality of the measurement setup by turning the random number generator off.

Note that Coco verifies ASIC netlists of masked implementations and identifies problematic wires where leaks might occur. The exact structure of this netlist must be reflected on the final FPGA layout to make concrete security statements, which is why we cannot simply synthesize the hardware design to the FPGA. The synthesis process will possibly merge multiple ASIC gates into a single lookup table (LUT) on the FPGA, and the original netlist structure will not be preserved. Consequently, one might see artifacts in the measurements stemming from this merging process, e.g. because the strict separation of shares is lost in the translation process [Cnu+17]. Therefore, we must ensure to map each gate in the verified ASIC netlist to a functionally equivalent FPGA LUT, in order to preserve the original netlist structure as good as possible. We achieve this by mapping each ASIC gate to a LUT with 2 inputs and one output, by putting a `dont_touch = "true"` on every gate/wire in the netlist.

5. Application to Masked Software Implementations

In this section we discuss how Coco can be used to identify leaks in arithmetically masked RISC-V assembly implementations. In the beginning, we outline the software verification setup. First, we focus on algebraic conversions, including Coron et al. [CGV14], Schneider et al. [Sch+19] for prime moduli and Goubin-

A2B/B2A [Gou01], for which we point out several register overwrite leaks. We discuss the table-based conversion algorithms of Debraize [Deb12] and Beirendonck et al. [BDV21], and explain how table lookups can be formally verified from a probing-security perspective. To conclude the section, we verify the masked ARX-based schemes Speck 32/64 and Alzette.

5.1. Software Verification Setup

Potential leaks in masked software are either caused by flaws in the algorithmic design, or due to microarchitectural side-effects of the processor's hardware. Flaws in the algorithmic design are mainly attributed to non-uniform sharings of intermediate variables, accidental combinations of masks, or transition leakage caused by register overwrites. However, even if such issues are taken into account there is still no guarantee that such an algorithm, once implemented for a specific processor, will be free of leaks. For example, a recent work by Gigerl et al. [Gig+21] has analyzed the RISC-V Ibex core in terms of architecture side-effects for masked software, and has pointed out multiple additional potential sources of leakage due to the design of the register file, the SRAM, the ALUs, and the load-store unit. They created a *secured* Ibex[2] that incorporates some relatively cheap hardware fixes that mostly eliminate glitch-related issues that are otherwise difficult to deal with purely on software-level.

For the purpose of this paper we are not so much interested into further netlist modifications, but rather focus on potential flaws in the algorithmic design of masked software implementations. We use their *secured* Ibex core as a reference platform that comes with a concrete list of hardware side-effects that do or do not need to be taken into consideration in software, thus allowing for an even playing field when evaluating and comparing different masked software implementations of A2B/B2A conversion algorithms. More specifically, the certain common microarchitectural leakages do not need to be addressed in software because the *secured* Ibex already has appropriate fixes on netlist-level. These fixes include:

- A glitch-resistant register file which allows to read and write shares without combination, as long as the respective software constraints are met
- No hidden registers or always-active computation units
- A glitch-resistant model of the SRAM (similar to the register file)

For more details on these fixes, we refer to the work of Gigerl et al. [Gig+21]. When a masked assembly implementation is executed by the *secured* Ibex and the software constraints are met, the leakages which are left are primarily register/memory overwrites and leaks caused by algorithmic flaws. The results of the following analysis can therefore be ported to any other microprocessor, as long as the respective device-specific fixes against these leaks, either in hardware or in software, are implemented.

[2] https://github.com/IAIK/coco-ibex

The synthesis process will transform the adder, which lies in the ALU of the *secured* IBEX, to a set of logic gates. Theoretically, each gate is each assigned a correlation set during verification, which is very time-consuming. We wrap up the addition into a custom `adder` "gate" instead of splitting it up, which means only the output wires of the adder must be assigned correlation sets. In order to achieve this, we identify the addition in the CPU design before synthesis (which is trivial), move it into a distinct module, and apply `keep_hierarchy` on this module, which results in a single `adder` gate on netlist level. In case of the *secured* IBEX core, the `adder` gate is represented by 2×32-bit inputs, and creates a 33-bit output, for which we can compute the correlation set quite efficiently using Equation 5 and Equation 6. Without this optimization, the synthesizer would split the adder up into individual logic gates and one would check the correlation of each of these gates individually. Consequently, especially verification in transient mode would then not be possible in a feasible time frame [3]. It is important to note that this does not affect the soundness guarantees of our approach because the correlation sets computed for the outputs of the `adder` gates are identical to the correlation sets of an adder which is split up.

5.2. Verification of Algebraic Share Conversions

Coron et al.-A2B/B2A [CGV14]

In Section 4 we discuss the verification of Coron et al. [CGV14] conversion algorithms in hardware. When verified on a CPU netlist, the algorithm in general behaves very similar. As shown in Table 1, we implement Coron et al.-A2B and -B2A in software and verify it successfully. We provide 16-bit A2B and B2A implementations which we verify in all three modes. Additionally, we implement 4-bit first- and second-order implementations, which can also successfully be verified with our approach. Compared to the results of Section 4, where we verify a 34-cycle implementation in 3 min, we can verify the respective software implementation (~1000 cycles) in 20 min, which shows the efficiency of our tool. Interestingly, both QMVERIF and `LeakageVerif` have to fall back to exhaustive enumeration when verifying Coron et al.-B2A, while the other direction (A2B) is possible [MPH21].

Schneider et al.-B2A [Sch+19]

Various A2Bs/B2As work with power-of-two moduli exclusively, while many lattice-based constructions require a prime modulus. To address this issue, one can first transform the shares from $\mathbb{F}_q$ to $\mathbb{F}_{2^k}$, and then apply conversion algorithms working with power-of-two moduli [Ode+18]. Another possibility

[3] Runtime of a few hours for a single 32-bit addition

is to directly adapt a $\mathbb{F}_{2^k}$-conversion algorithm to work in $\mathbb{F}_q$. We investigate Schneider et al.-B2A [Sch+19], which is an adaption of Coron et al.-B2A, and was initially proposed to build a masked binomial sampler. We sketch the algorithm in Appendix E. We construct a first-order implementation of Schneider et al.-B2A with $q = 257$ and 4-bit input shares.

During conversion, Schneider et al.-B2A heavily uses reductions mod q, which are usually not implemented using the processor's `mod` instruction due to the instruction's large runtime overhead. Instead, many practical implementations use efficient reduction methods like Montgomery [Mon85] or Barret [Bar86] in combination with lazy reduction, i.e., skipping reductions as long as intermediate values are guaranteed to fit inside 32-bit words (on 32-bit architectures) [BKS19]. For our implementation, we eliminate all reductions except the very last at the end of the algorithm, where we stick to Barret reduction. These tricks not only significantly improve runtime but also reduce the verification runtime drastically, since mod operations create very complex dependencies between individual bits of a share. In this setting, we want to point out and interesting pitfall that should be avoided when using lazy reduction techniques in the context of masking. For example, in order to convert the Boolean shares b_0 and b_1, Schneider et al.-B2A first generates a random number $E_0 \in \mathbb{F}_q$, and then computes $E_1 = ((b_1 - E_0 \mod q) - 2 \cdot ((b_1 - E_0 \mod q) \cdot b_0)) \mod q$. If one now lazily skips the reductions, the upper bits of E_1 will not be masked anymore due to the smaller bit width of E_0 . To mitigate this potential pitfall on 32-bit architectures, one could simply always use 32-bit words of randomness whenever mask refreshing is required. Other than that, the verification points out no issues in the algorithm.

Goubin-A2B/B2A [Gou01]

Goubin's algorithms [Gou01] fix one output share, while the other is computed accordingly in order to derive a correct arithmetic or Boolean sharing. Goubin-B2A fixes $a_1 = b_1$ and then applies the recursive rule $a_0 = (b \oplus b_1) - b_1$ to compute the second share, while the respective A2B fixes $b_1 = a_1$ and computes b_0 by recursion instead. Goubin-B2A remains popular due to its efficiency ($\mathcal{O}(1)$), while the A2B conversion is more costly ($\mathcal{O}(n)$ for n bits [CGV14]). In Appendix G we outline both algorithms. We can successfully verify the security of the B2A conversion in the stable, transition, and transient case. However, we encounter several issues with the A2B conversion that we now describe in more detail. To the best of our knowledge, these findings have not been reported yet.

First, Goubin-A2B introduces several problems regarding insecure register overwrites even if we ensure that our implementation uses dedicated registers for each of the variables proposed in the original algorithm. The algorithm uses an intermediate Y, which is initialized with a random variable and overwritten several times during the computation. Each of these overwrites leaks the XOR between the old (Y_{old}) and the new (Y_{new}) value. The verifier points out two situations during the computation in which $Y_{\text{old}} \oplus Y_{\text{new}}$ reveals information about

the native value a. This issue can however be fixed easily be ensuring that different registers are used for every re-assignment of Y. In Appendix G we give a detailed calculation of the issue.

Second, the verifier indicates another leak during the computation, which is however not a practical problem, but a false positive as already mentioned in Section 3.2. In the following, we want to briefly highlight the circumstances and give the exact calculation in Appendix G. During the computation, an attacker can probe the following 1-bit expression: $(Y^{(0)} \wedge (Y^{(0)} \oplus a_1^{(0)})) \oplus (a_1^{(0)} \wedge (Y^{(0)} \oplus a_0^{(0)}))$, with $Y^{(0)}$ being random. The exact Fourier expansion of this expression does not contain a single term which depends on both $a_0^{(0)}$ and $a_1^{(0)}$ alone, but only in connection with $Y^{(0)}$, and is therefore properly masked. However, the verifier works with approximated correlation sets, which contain a subset $\{a_0^{(0)}, a_1^{(0)}\}$ where the random value $Y^{(0)}$ is not contained, and therefore represents a leak. According to [MPH21], both QMVerif and `LeakageVerif` also fail to verify Goubin-A2B correctly because their tools produce false positives. Unfortunately, they do not discuss the exact issue, and therefore we were not able to make further investigations.

5.3. Verification of Table-based Share Conversions

Besides algebraic approaches, several A2Bs utilize lookups into pre-computed tables, such as the ones from Debraize [Deb12] and Beirendonck et al. [BDV21]. A table lookup represents a data-dependent memory access, i.e., an operation that loads data from a memory address that is data-dependent. Coco was mostly intended to verify symmetric cryptography, where table lookups are not common and have therefore not been considered. However, our study shows that the verification approach can be successfully applied under specific conditions, which are fortunately fulfilled by all table-based A2Bs that we are aware of.

First, it must be possible to compute all entries in the table with a single unique function $f(i)$, which depends only on the table index i and constants. This ensures that every table entry is assigned the same label during the verification independently of the address. For example, Debraize-A2B uses $f(i) = i + r + p \oplus (p||r)$ for initially generated random values ("constants") r and p. Second, the evaluation platform must guarantee constant-time memory accesses, i.e., memory accesses always require the same amount of cycles independently of the memory address. For example, the original Ibex core fetches multiple memory locations in case of a misaligned memory access, and therefore requires more cycles compared to an aligned memory access. Therefore, we simply disable the *secured* Ibex core's ability to perform misaligned memory accesses, which represents a quite reasonable modification for verification purposes since constant-time is anyway a desired property of cryptographic implementations.

Table 2.: Verification results for masked software: ✔(no issues were found), (✔) (algorithmically secure, but potential problems like table-lookups), ✖(algorithmically insecure implementations), ✖(false positive).

Algorithm	Input shares	Runtime (cycles)	Verification result/runtime					
			Stable		Transitions		Transient	
A2Bs								
Coron et al. [CGV14]	4 bit	225	✔	41 s	✔	67 s	✔	3 min
Coron et al. [CGV14]	16 bit	984	✔	4 min	✔	5 min	✔	16 min
Coron et al. [CGV14] (2nd order)	4 bit	1240	✔	3.8 min	✔	6 min	✔	20 min
Debraize [Deb12] ▦	4 bit ($n = 2$, $k = 2$)	140	✖	35 s	-		-	
Debraize [Deb12] ▦	16 bit ($n = 4$, $k = 4$)	450	✖	118 s	-		-	
Beirendonck et al.-fixed-Debraize [BDV21] ▦	4 bit ($n = 2$, $k = 2$)	180	(✔)	38 s	(✔)	48 s	-	
Beirendonck et al.-Dual-Lookup [BDV21] ▦	4 bit ($n = 2$, $k = 2$)	105	(✔)	28 s	(✔)	30 s	-	
Goubin [Gou01]	16 bit	170	✖	37 s	✖		-	
B2As								
Goubin [Gou01]	16 bit	23	✔	5 s	✔	8 s	✔	19 s
Coron et al. [CGV14]	4 bit	650	✔	6 min	✔	4 min	✔	11 min
Coron et al. [CGV14]	16 bit	2475	✔	11 min	✔	16 min	✔	38 min
Schneider et al. [Sch+19], without final mod instruction	4 bit, ($q = 257, \log_2 q = 9$)	400	(✔)	2 min	(✔)	21 min	-	
ARX-based schemes								
Speck 32/64 1 round	6 × 16 bit	1465	✔	6 min	✔	13 min	✔	5.13 h
Alzette 1 round	2 × 32 bit	3082	✔	29 min	✔	2.48 h	✔	27 h

5.4. Application to Table-based Conversion Algorithms

Table-based A2Bs usually take over one Boolean share from the arithmetic domain and derive the second by computing $b_0 = (a_0 + a_1) \oplus a_1$. From a masking perspective, $(a_0 + a_1)$ leaks the native value a, which is prevented using a precomputed look-up table which stores $(a_0 + r) \oplus r$ for a fixed r [BDV21]. However, generating the table for each possible value of a_0 is not efficient.

Debraize-A2B [Deb12]

In 2012, Debraize [Deb12] suggests to split up a_0 into n parts of k bits each, and precompute a table entry for each of the 2^k possible values. The actual conversion is performed by iterating over the n parts of a_0 and converting each part individually by performing a table lookup to the precomputed table. The table returns (a) the transformed part of a_0 into the Boolean domain, and (b) the one-bit carry that is Boolean masked and has to be considered in the next iteration. We sketch the algorithm in Appendix F. We implement Debraize-A2B with $n = 2, k = 2$ as well as $n = 4, k = 4$ and verify its execution as shown in Table 2. Two leaks are already reported in the stable verification mode (indicated by ✖), which points towards algorithmic errors.

First, the verifier reports a leak when performing the table lookup due to a combination of the (share-dependent) address bits and the memory content. On gate level, a table lookup using 32-bit addresses is realized using an equality comparator, which itself consists of 32 XNOR gates, whose output is combined by a single AND gate [Man82]. In case of equality, the AND gate outputs 1, and 0 otherwise. This information is finally used to decide whether to read data from a specific location. We give an example of an equality comparator in Appendix H. When performing the table lookup in Debraize-A2B in the first iteration, the address bits depend on both arithmetic shares, a random value r and a random bit β, which represents an intermediate result of the transformation. The content of the precomputed lookup-table is determined by r and β. Combining both values cancels out the random values r and β, and the attacker can probe an expression depending on the native value a. One can argue that an SRAM module is constructed in a way such that the address and the memory cell content will never be combined. However, in bigger CPUs, the memory access logic is much more complicated and might contain buffers or caches, which employ such an addressing mechanism. For example, data caches usually require the computation of a tag based on the address, and compare this tag to the one in the cache.

Second, the verifier points out that the value obtained from the lookup-table in the first iteration is not uniformly distributed, although used as a mask in the algorithm. Beirendonck et al. [BDV21] already report the issue in their work, which was found by empirical measurements, and provide a theoretical analysis afterwards. We want to emphasize that another advantage of our verification approach is the fast discovery of such bugs, which happens in 35 s and 118 s

according to Table 2 in this case, which is much quicker than empirical/theoretical evaluations. Coco reports the leaking cycle and netlist gate immediately and therefore one does not need to carry out a laborious empirical analysis.

Beirendonck et al.-A2Bs [BDV21]

In their work, Beirendonck et al. [BDV21] propose two new secure table-based A2Bs, Beirendonck et al.-fixed-Debraize A2B (a secured version of Debraize-A2B) and Beirendonck et al.-Dual-Lookup A2B (an efficient version of Beirendonck et al.-fixed-Debraize A2B). We verify both algorithms by choosing parameters $n = 2, k = 2$. As shown in Table 2, table lookups cause a similar leak as we already discussed for Debraize-A2B. Since the issue however strongly depends on the underlying microarchitecture, and no further issues were found, we mark it with (✔) in the table.

5.5. Application to ARX-based Constructions

The ARX (Addition-Rotation-XOR) design principle has been used for several well-known symmetric cryptographic constructions like the block cipher Speck [Bea+13], the stream cipher ChaCha [Ber08], or the hash function SHA-256 [Nat02]. We focus on first-order implementations of a single round of Speck 32/64 [Bea+13], and the 64-bit ARX-based S-box Alzette [Bei+20]. Alzette is a central building block of Sparkle, which is currently one of the finalists of the NIST LWC Standardization Process [Tur+21]. Masking these implementations requires both Boolean masking (for the Rotation and XOR) and arithmetic masking (for the addition). One option is to apply an algorithm like SecAdd, which implements modular addition directly on Boolean shares [CGV14; DGC17; KRJ14; SMG15]. Another possibility is to first convert the Boolean shares to arithmetic shares, then perform the addition on arithmetic shares, and convert the shares back to the Boolean domain. In our implementation, we choose the second option using Goubin-B2A before each addition, perform the addition on arithmetic shares, and switch back to the Boolean domain using Coron et al.-A2B. We are able to verify algorithmic security in under 30 minutes for both schemes (stable mode). For the transient mode, the verification requires several hours, which is mostly spent by solving the SAT equation, and therefore offers several possibilities for further optimization.

6. Conclusion

In this paper, we presented an approach for the formal verification of masked software and hardware implementations, which supports both arithmetic and Boolean masking schemes of any order. On the hardware side, we show that glitches may cause issues in the context of masking for a straightforward implementation of Coron et al.-A2B. We demonstrate that this issue exists in practice using

empirical measurements. On the software side, we first analyze algebraic share conversions, report a previously unknown register transition issue in Goubin-A2B and provide new insights on the security of lazy reduction, a popular optimization technique in PQC. Second, we discuss table-based conversions and demonstrate that table lookups might not be secure due to architectural side-effects. Last but not least, we underline the scalability of our approach by applying it to entire round functions of masked ARX-based ciphers.

Acknowledgements

This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". Additionally, this project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Appendix A. Iterative and unrolled circuits

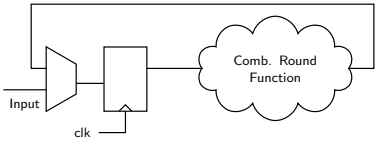


Figure 3.: Iterative circuit [Bha+10]

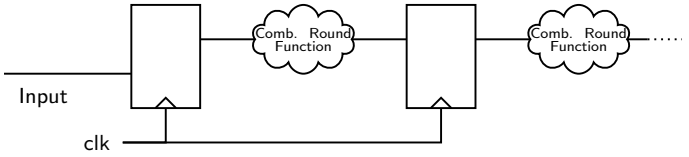


Figure 4.: Unrolled circuit [Bha+10]

Appendix B. Fourier Expansion of the Arithmetic Addition

Recall the Fourier expansion of the AND, OR and XOR functions:

$$\begin{aligned}
\textsc{And} \quad & W(a \wedge b) = \frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b - \frac{1}{2}ab \\
\textsc{Or} \quad & W(a \vee b) = -\frac{1}{2} + \frac{1}{2}a + \frac{1}{2}b + \frac{1}{2}ab \\
\textsc{Xor} \quad & W(a \oplus b) = ab
\end{aligned}$$

Additionally, note that Fourier expansions represent Boolean functions as a polynomial over the real domain $\{1, -1\}$, where 1 represents False and -1 represents True. Consequently, monomials x^c with even exponents c evaluate to 1 in Fourier expansions. The Fourier expansion of the carry and sum can hence be expressed as:

$$\begin{aligned}
\textsc{Carry} \quad & W(c^{(j)}) = W((u^{(j)} \oplus u^{(j)}) \wedge c^{(j-1)}) \vee (u^{(j)} \wedge u^{(j)})) \\
& = -(0.25u^{(j)})^2(u^{(j)})^2c^{(j-1)} - 0.25(u^{(j)})^2(u^{(j)})^2 \\
& - 0.25(u^{(j)})^2u^{(j)}c^{(j-1)} - 0.25u^{(j)}(u^{(j)})^2c^{(j-1)} + (0.25u^{(j)})^2u^{(j)} \\
& + 0.25u^{(j)}(u^{(j)})^2 - 0.5u^{(j)}u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} \\
& + 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)} + 0.25u^{(j)} + 0.25c^{(j-1)} + 0.25 \\
& = 0.25c^{(j-1)} - 0.25 - 0.25u^{(j)}c^{(j-1)} - 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)} \\
& + 0.25u^{(j)} - 0.5u^{(j)}u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} + 0.25u^{(j)}c^{(j-1)} \\
& + 0.25u^{(j)} + 0.25u^{(j)} + 0.25c^{(j-1)} + 0.25 \\
& = 0.5c^{(j-1)} + 0.5u^{(j)} + 0.5u^{(j)} - 0.5u^{(j)}u^{(j)}c^{(j-1)} \\
& W(c[0]) = 1 \\
\textsc{Sum} \quad & W(sum^{(j)}) = W(W(u^{(j)} \oplus u^{(j)}) \oplus c^{(j)}) \\
& = W(u^{(j)}u^{(j)} \oplus c^{(j)}) \\
& = u^{(j)}u^{(j)}W(c^{(j)}) \\
& = 0.5u^{(j)}u^{(j)}c^{(j)} + 0.5u^{(j)} + 0.5u^{(j)} - 0.5c^{(j)}
\end{aligned}$$

Appendix C. Coron et al.-A2B

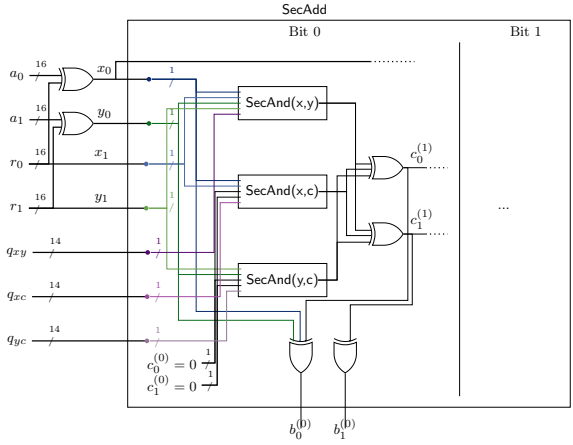


Figure 5.: Schematic image of Coron et al.-A2B [CGV14] when implemented in hardware. The arithmetic input shares a_0, a_1 are transformed into Boolean shares b_0, b_1. The carry computation happens in the SecAdd module, from which we draw the first part responsible for bits 0 of the final result.

Appendix D. Sanity Check Measurement Setup (RNG Off)

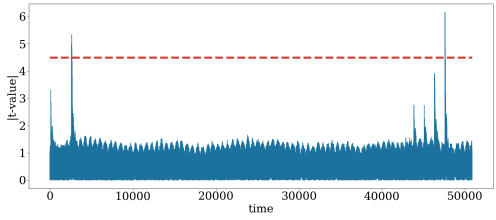


Figure 6.: T-test statistics of the fixed version of Coron et al.-A2B with 400 000 traces and RNG off.

Appendix E. Schneider et al.-B2A [Sch+19]

Algorithm 1 Schneider et al. B2A [Sch+19] (simplified for 1st order)

Input: k-bit shares b_0, b_1 such that $b = b_0 \oplus b_1$
Output: Shares $a_0, a_1 \in \mathbb{F}_q$ such that $b = a_0 + a_1 \mod q$

$b'_0 \leftarrow b_0^{(k-1)}$
$b'_1 \leftarrow b_1^{(k-1)}$
$a_0, a_1 \leftarrow \text{B2A\_Bit}(b'_0, b'_1)$
$R \xleftarrow{\$} \mathcal{R}_q$
$a_0 \leftarrow (a_0 + R) \mod q$
$a_1 \leftarrow (a_1 - R) \mod q$
for $j = 2$ to $k - 1$ **do**
  $b'_0 \leftarrow b_0^{(k-j)}$
  $b'_1 \leftarrow b_1^{(k-j)}$
  $C_0, C_1 \leftarrow \text{B2A\_Bit}(b'_0, b'_1)$
  $R \xleftarrow{\$} \mathcal{R}_q$
  $C_0 \leftarrow (C_0 + R) \mod q$
  $a_0 \leftarrow ((a_0 << 1) + C_0) \mod q$
  $C_1 \leftarrow (C_1 - R) \mod q$
  $a_1 \leftarrow ((a_1 << 1) + C_1) \mod q$
end for
return a_0, a_1

Algorithm 2 Schneider et al. B2A_Bit [Sch+19] (simplified for 1st order)

Input: 1-bit shares b'_0, b'_1 such that $b = b'_0 \oplus b'_1$
Output: E_0, E_1 such that $E_0 + E_1 = b \mod q$

$E_0 \xleftarrow{\$} \mathcal{R}_q$
$E_1 \leftarrow b'_1 - E_0 \mod q$
$E_1 \leftarrow E_1 - 2 \cdot (E_1 \cdot b'_0) \mod q$
$E_0 \leftarrow E_0 - 2 \cdot (E_0 \cdot b'_1) \mod q$
$E_1 \leftarrow E_1 + b'_1 \mod q$
return E_1, E_0

Appendix F. Debraize-A2B

Algorithm 3 Table T generation [Deb12]

Input: k
Output: Conversion table T, random variables r, ρ

$r \leftarrow \mathcal{U}(0,1)^k$
$\rho \leftarrow \mathcal{U}(0,1)$
for $i \leftarrow 0$ to $2^k - 1$ **do**
  $T[\rho||i] \leftarrow (i + r) \oplus (\rho||r)$
  $T[(\rho \oplus 1)||i] \leftarrow (i + r + 1) \oplus (\rho||r)$
end for
return T, r, ρ

Algorithm 4 Debraize-A2B [Deb12]

Input: $(n \cdot k)$-bit shares a_0, a_1 such that $a = a_0 + a_1 \mod 2^{(n \cdot k)}, T, r, \rho$
Output: $(n \cdot k)$-bit shares b_0, b_1 such that $a = b_0 \oplus b_1$

$a_0 \leftarrow a_0 - (r||...||r||...||r) \mod 2^{n \cdot k}$
$\beta \leftarrow \rho$
for $i \leftarrow 0$ to $n - 1$ **do**
  Split a_0 into $(a_{0h}||a_{0l})$, split a_1 into $(a_{1h}||a_{1l})$
  $a_0 \leftarrow a_0 + a_{1l} \mod 2^{(n-i) \cdot k}$
  $\beta||x'_i \leftarrow T[\beta||a_{0l}]$
  $x'_i \leftarrow x'_i \oplus a_{1l}$
  $a_0 \leftarrow a_{0h}, a_1 \leftarrow a_{1h}$
end for
$b_0 = (x'_0||...||x'_i||...||x'_{n-1}) \oplus (r||...||r||...||r)$
$b_1 = a_1$
return b_0, b_1

Appendix G. Goubin [Gou01]

Algorithm 5 Goubin-A2B [Gou01]

Input: n-bit shares a_0, a_1 such that $a = a_0 + a_1 \mod 2^n$
Output: n-bit shares b_0, b_1 such that $a = b_0 \oplus b_1$

1: $Y \leftarrow \mathcal{U}(0,1)^n$
2: $T \leftarrow 2Y$
3: $b_0 \leftarrow Y \oplus a_1$
4: $\Omega \leftarrow Y \wedge b_0$
5: $b_0 \leftarrow T \oplus a_0$
6: $Y \leftarrow Y \oplus b_0$
7: $Y \leftarrow Y \wedge a_1$
8: $\Omega \leftarrow \Omega \oplus Y$
9: $Y \leftarrow T \wedge a_0$
10: $\Omega \leftarrow \Omega \oplus Y$
11: **for** $i \leftarrow 0$ to $n-1$ **do**
12: $\quad Y \leftarrow T \wedge a_1$
13: $\quad Y \leftarrow Y \oplus \Omega$
14: $\quad T \leftarrow T \wedge a_0$
15: $\quad Y \leftarrow Y \oplus T$
16: $\quad T \leftarrow 2Y$
17: **end for**
18: $b_0 \leftarrow b_0 \oplus T$
19: $b_1 \leftarrow a_1$
20: **return** b_0, b_1

Algorithm 6 Goubin-B2A [Gou01]

Input: n-bit shares b_0, b_1 such that $a = b_0 \oplus b_1$
Output: n-bit shares a_0, a_1 such that $a = a_0 + a_1 \mod 2^n$

1: $Y \leftarrow \mathcal{U}(0,1)^n$
2: $T \leftarrow b_0 \oplus Y$
3: $T \leftarrow T - Y$
4: $T \leftarrow T \oplus b_0$
5: $Y \leftarrow Y \oplus b_1$
6: $a_0 \leftarrow b_0 \oplus Y$
7: $a_0 \leftarrow a_0 - Y$
8: $a_0 \leftarrow a_0 \oplus T$
9: $a_1 \leftarrow b_1$
10: **return** a_0, a_1

Overwrite leakages

In line 9 of the algorithm, the attacker probes the re-assignment of Y:

$$\begin{aligned} Y_{\text{old}} &= Y_{\text{line 6}} \wedge a_1 \\ &= (Y_{\text{line 1}} \oplus b_{0\text{line 5}}) \wedge a_1 \\ &= (Y_{\text{line 1}} \oplus (T \oplus a_0)) \wedge a_1 \\ Y_{\text{new}} &= T \wedge a_0 \\ Y_{\text{old}} \oplus Y_{\text{new}} &= ((Y_{\text{line 1}} \oplus (T \oplus a_0)) \wedge a_1) \oplus (T \wedge a_0) \\ &= (a_0 \wedge a_1) \oplus (a_0 \wedge T) \oplus (a_1 \wedge Y) \end{aligned}$$

Hence, for every bit $>= 0$, this expression will correlate with native value a.

Another similar situation occurs in Figure 12 where $Y_{\text{old}} = T \wedge a_0$ is overwritten by $Y_{\text{new}} = T \wedge a_1$ in the first loop iteration.

False positive in Goubin-A2B

Assume the attacker probes the expression $\Omega \oplus Y_{\text{line 9}}$ in line 10, which is $(Y^{(0)} \wedge (Y^{(0)} \oplus a_1^{(0)})) \oplus (a_1^{(0)} \wedge (Y^{(0)} \oplus a_0^{(0)}))$. For reasons of readability, we omit to indicate that we always refer to the LSB, i.e., skip $^{(0)}$.

Exact Fourier expansion

$$
\begin{aligned}
W((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= ? \\
W(Y \oplus a_0) &= Y a_0 \\
W(Y \oplus a_1) &= Y a_1 \\
W(Y \wedge (Y \oplus a_1)) &= -0.5\, Y^2 a_1 + 0.5\, Y a_1 + 0.5\, Y + 0.5 \\
&= -0.5\, a_1 + 0.5\, Y a_1 + 0.5\, Y + 0.5 \\
W((Y \oplus a_0) \wedge a_1)) &= -0.5\, Y a_0 a_1 + 0.5\, Y a_0 + 0.5\, a_1 + 0.5 \\
W((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= -0.25\, Y^2 a_0 a_1^2 + 0.25\, Y a_0 a_1^2 + 0.25\, Y^2 a_0 - 0.5\, Y a_0 a_1 \\
&\quad + 0.25\, Y a_1^2 + 0.25\, Y a_0 + 0.50\, Y a_1 - 0.25\, a_1^2 + 0.25\, Y + 0.2 \\
&= -0.25\, a_0 + 0.25\, Y a_0 + 0.25\, a_0 - 0.5\, Y a_0 a_1 \\
&\quad + 0.25\, Y + 0.25\, Y a_0 + 0.50\, Y a_1 - 0.25 + 0.25\, Y + 0.25
\end{aligned}
$$

Approximated Fourier expansion

$$
\begin{aligned}
\mathcal{C}((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= ? \\
\mathcal{C}(Y \oplus a_0) &= \{\{Y, a_0\}\} \\
\mathcal{C}(Y \oplus a_1) &= \{\{Y, a_1\}\} \\
\mathcal{C}(Y \wedge (Y \oplus a_1)) &= \{\{1\}, \{Y\}, \{Y, a_1\}, \{a_1\}\} \\
\mathcal{C}((Y \oplus a_0) \wedge a_1)) &= \{\{1\}, \{Y, a_0\}, \{a_1\}, \{Y, a_0, a_1\}\} \\
\mathcal{C}((Y \wedge (Y \oplus a_1)) \oplus ((Y \oplus a_0) \wedge a_1)) &= \mathcal{C}((Y \oplus a_0) \wedge a_1)) \otimes \mathcal{C}(Y \wedge (Y \oplus a_1))\} \\
&= \{\{1\}, ...\{Y^2, a_0, a_1\}, ...\}
\end{aligned}
$$

Note: $Y^2 = 1$ because in Fourier expression each element is either 1 (False) or -1 (True).

Appendix H. Table lookup on gate-level

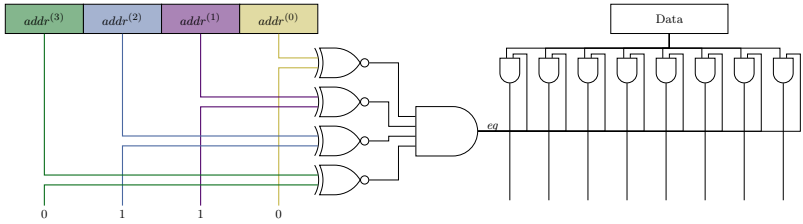


Figure 7.: Example of table lookup including equality comparator on gate-level with 4-bit addresses and 8-bit data words. The address *addr* is compared to the constant address of the SRAM cell ($(0110)_b$). If both values are equal, the resulting 1-bit signal *eq* is 1, and 0 otherwise. *eq* is further used to decide whether the respective data word should be read or not.

References

[AFM17] Alexandre Adomnicai, Jacques J. A. Fournier, and Laurent Masson. "Bricklayer Attack: A Side-Channel Analysis on the ChaCha Quarter Round". In: *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*. Ed. by Arpita Patra and Nigel P. Smart. Vol. 10698. Lecture Notes in Computer Science. Springer, 2017, pp. 65–84. DOI: 10.1007/978-3-319-71667-1_4. URL: https://doi.org/10.1007/978-3-319-71667-1%5C_4.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 457–485.

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings,*

Part II. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 354–384.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[Bar+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 189–228.

[Bar86] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 311–323. DOI: 10.1007/3-540-47721-7_24. URL: https://doi.org/10.1007/3-540-47721-7%5C_24.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. "Improved High-Order Conversion From Boolean to Arithmetic Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 22–45.

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. "Analysis and Comparison of Table-based Arithmetic to Boolean Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 275–297.

[Bea+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. "The SIMON and SPECK Families of Lightweight Block Ciphers". In: *IACR Cryptol. ePrint Arch.* (2013), p. 404.

[Bei+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Alzette: A 64-Bit ARX-box - (Feat. CRAX and TRAX)". In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12172. Lecture Notes in Computer Science. Springer, 2020, pp. 419–448.

[Ber08] Daniel J Bernstein. "ChaCha, a variant of Salsa20". In: *Workshop record of SASC*. Vol. 8. 2008, pp. 3–5.

[Bha+10] Shivam Bhasin, Sylvain Guilley, Laurent Sauvage, and Jean-Luc Danger. "Unrolling Cryptographic Circuits: A Simple Countermeasure Against Side-Channel Attacks". In: *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*. Ed. by Josef Pieprzyk. Vol. 5985. Lecture Notes in Computer Science. Springer, 2010, pp. 195–207.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4". In: *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*. Ed. by Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi. Vol. 11627. Lecture Notes in Computer Science. Springer, 2019, pp. 209–228. DOI: 10.1007/978-3-030-23696-0_11. URL: https://doi.org/10.1007/978-3-030-23696-0%5C_11.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Bos+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. "Masking Kyber: First- and Higher-Order Implementations". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 173–214.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 188–205.

[Che+15] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. "Masking Large Keys in Hardware: A Masked Implementation of McEliece". In: *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*. Ed. by Orr Dunkelman

and Liam Keliher. Vol. 9566. Lecture Notes in Computer Science. Springer, 2015, pp. 293–309.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[Cnu+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. "Does Coupling Affect the Security of Masked Implementations?" In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 1–18. DOI: 10.1007/978-3-319-64647-3_1. URL: https://doi.org/10.1007/978-3-319-64647-3%5C_1.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 69–81.

[Cor+13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Springer, 2013, pp. 410–424. DOI: 10.1007/978-3-662-43933-3_21. URL: https://doi.org/10.1007/978-3-662-43933-3%5C_21.

[Cor+15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*. Ed. by Gregor Leander. Vol. 9054. Lecture Notes in Computer Science. Springer, 2015, pp. 130–149. DOI: 10.1007/978-3-662-48116-5_7. URL: https://doi.org/10.1007/978-3-662-48116-5%5C_7.

[Cor+22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. "High-order Table-based Conversion Algorithms and Masking Lattice-based Encryption". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.2 (2022), pp. 1–40.

[Cor17] Jean-Sébastien Coron. "High-Order Conversion from Boolean to Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 93–114.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 89–97.

[Deb12] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 107–121.

[DGC17] Daniel Dinu, Johann Großschädl, and Yann Le Corre. "Efficient Masking of ARX-Based Block Ciphers Using Carry-Save Addition on Boolean Shares". In: *Information Security - 20th International Conference, ISC 2017, Ho Chi Minh City, Vietnam, November 22-24, 2017, Proceedings*. Ed. by Phong Q. Nguyen and Jianying Zhou. Vol. 10599. Lecture Notes in Computer Science. Springer, 2017, pp. 39–57.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 537–554.

[Fri+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 414–460.

[Gao+19a] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. "Quantitative Verification of Masked Arithmetic Programs Against Side-Channel Attacks". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 155–173. DOI: 10.1007/978-3-030-17462-0_9. URL: https://doi.org/10.1007/978-3-030-17462-0%5C_9.

[Gao+19b] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. "Verifying and Quantifying Side-channel Resistance of Masked Software Implementations". In: *ACM Trans. Softw. Eng. Methodol.* 28.3 (2019), 16:1–16:32. DOI: 10.1145/3330392. URL: https://doi.org/10.1145/3330392.

[Gao+20] Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. "A Hybrid Approach to Formal Verification of Higher-Order Masked Arithmetic Programs". In: *CoRR* abs/2006.09171 (2020).

[Gao20] Pengfei Gao. "Formal Verification of Masking Countermeasures for Arithmetic Programs". In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 1385–1387.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 1–21.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A testing methodology for side-channel resistance validation". In: *NIST Non-Invasive Attack Testing Workshop*. 2011.

[Gou01] Louis Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 3–15.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors". In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GR19] François Gérard and Mélissa Rossi. "An Efficient and Provable Masked Implementation of qTESLA". In: *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*. Ed. by Sonia Belaïd and Tim Güneysu. Vol. 11833. Lecture Notes in Computer Science. Springer, 2019, pp. 74–91. DOI: 10.1007/978-3-030-42068-0_5. URL: https://doi.org/10.1007/978-3-030-42068-0%5C_5.

[HB21] Vedad Hadzic and Roderick Bloem. "COCOALMA: A Versatile Masking Verifier". In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.34727/2021/isbn.978-3-85448-046-4_9. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4%5C_9.

[HT19] Michael Hutter and Michael Tunstall. "Constant-time higher-order Boolean-to-arithmetic masking". In: *J. Cryptogr. Eng.* 9.2 (2019), pp. 173–184.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KRJ14] Mohamed Karroumi, Benjamin Richard, and Marc Joye. "Addition with Blinded Operands". In: *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 8622. Lecture Notes in Computer Science. Springer, 2014, pp. 41–55.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification". In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[Man82] M. Morris Mano. *Computer system architecture*. Prentice Hall, 1982. ISBN: 978-0-13-166637-5.

[Mon85] Peter L Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[Moo+19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. "Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 256–292.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. "Side-Channel Leakage of Masked CMOS Gates". In: *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. Springer, 2005, pp. 351–365.

[MPH21] Quentin L. Meunier, Etienne Pons, and Karine Heydemann. "LeakageVerif: Scalable and Efficient Leakage Verification in Symbolic Expressions". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1468. URL: https://eprint.iacr.org/2021/1468.

[Nat02] National Institute of Standards and Technology (NIST). *FIPS-180-2: Secure Hash Standard*. 2002. URL: http://www.itl.nist.gov/fipspubs/.

[NP04] Olaf Neiße and Jürgen Pulkus. "Switching Blindings with a View Towards IDEA". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 230–239.

[Ode+18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. "Practical CCA2-Secure and Masked Ring-LWE Implementation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 142–174.

[ODo14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5.

[QS01] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards". In: *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. Ed. by Isabelle Attali and Thomas P. Jensen. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001, pp. 200–210.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[Sch+19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. Lecture Notes in Computer Science. Springer, 2019, pp. 534–564.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. "Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware". In: *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*. Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. Lecture Notes in Computer Science. Springer, 2015, pp. 559–578.

[Tur+21] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, ÇagdaËs Çalık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. *Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process*. Tech. rep. Tech. rep. https://doi.org/10.6028/NIST. IR. 8369. Gaithersburg, MD, USA ..., 2021.

8

Security Aspects of Masking on FPGAs

Publication Data. Barbara Gigerl, Kevin Pretterhofer, and Stefan Mangard. "Security Aspects of Masking on FPGAs". In: *HOST*. 2024.

Contribution. The author of this thesis contributed to the concepts, performed all the experiments, and wrote the text for the paper. The verification tool was created by Kevin Pretterhofer in the context of his master thesis, which was supervised by the author of this thesis.

Security Aspects of Masking on FPGAs

Barbara Gigerl[1], Kevin Pretterhofer[1], Stefan Mangard[1]

[1] Graz University of Technology

Abstract Many IoT and automotive use cases employ cryptographic hardware implementations, which are susceptible to physical attacks such as power analysis. Masking is a popular approach to protect against these attacks on algorithmic level. In general, a masked hardware implementation must be secure in theory, but also in practice. The practical security of masked ASIC designs has been analyzed thoroughly in literature, especially regarding physical defaults such as glitches and transitions, and optimizations performed during synthesis. Besides ASICs, FPGAs are often used to implement masked hardware designs, which utilize reconfigurable lookup tables (LUTs) instead of binary logic gates to implement arbitrary logic functions. Due to their different structure, FPGAs apply different synthesis flows and optimizations, whose effects on the security of masked implementations have not been investigated yet.

In this work, we present a case study of leakage sources in masked hardware implementations on FPGAs. We investigate several implementations that are formally proven to be secure even in the presence of glitches when implemented as an ASIC but show leakage when running them on an FPGA. We demonstrate in several practical experiments that this leakage is caused by optimizations performed during synthesis, such as LUT combining or register retiming. In order to identify such leaks, we present Fenix, the first tool to formally verify any-order masked hardware implementations directly on FPGA netlists. Fenix takes glitches into account and automatically localizes the leaking wire in case of an insecure design. We demonstrate the practicality of our tool using several masked hardware implementations, including masked multiplication gadgets, a 2nd-order Keccak S-box, and a full Ascon round.

1. Introduction

Embedded devices have become an essential part of many IoT, automotive, and industrial applications, where they unavoidably get in touch with sensitive information. One key aspect is therefore privacy, which raises the need for strong cryptographic primitives that are able to withstand both theoretical (mathematical) and physical attacks. Physical attacks assume that the adversary has direct access to the device and can observe information about the device during the computation. A famous example is Differential Power Analysis (DPA) [KJJ99], which works by monitoring the power consumption of a cryptographic device

during computation. The power consumption correlates with the processed sensitive data, such as the encryption key, which can then be extracted using statistical tools. To defeat such attacks, implementations employ the masking countermeasure [Cha+99; GP99; ISW03], which splits each sensitive variable into $d + 1$ random shares and performs the cryptographic operations on these shares instead. During the computation, the power consumption of the device at a given point in time does not correlate with the unshared sensitive value, but with at most one share.

The masking countermeasure can be applied to cryptographic hardware implementations, which are realized using an ASIC (Application-Specific Integrated Circuit) or an FPGA (Field-Programmable Gate Array). While ASICs are custom circuits designed for a specific purpose, FPGAs can be configured many times with a design using configurable logic blocks (lookup tables). In recent years, FPGAs have become more and more important in the area of cryptography due to their short time-to-market and reconfigurability [Gün11; Pra+04].

The security of a masked cipher can formally be proven on algorithmic level by analyzing an abstract description of the protected cipher implementation. However, even if the masking scheme is theoretically secure, a concrete implementation of the scheme, such as a hardware implementation running on an FPGA, might still be insecure. Hardware-related physical side-effects such as glitches and transitions, which are not part of the theoretical model, can still lead to the unintentional combination of shares and therefore leak the sensitive value [Cas+21; CS20; GMK16; ISW03; NRR06; Rep+15; Tri03]. Additionally, the synthesis process, which transforms an HDL design into a gate-level netlist, may introduce leaks by performing optimizations such as reordering operations [Blo+18; Roy+15; SSM21].

Consequently, the verification of masked implementations has gained a lot of attention recently. In general, there are two approaches to determine whether a masked implementation is secure in practice. The first option is to perform empirical verification by fabricating the design as an ASIC or porting it to an FPGA, recording power traces, and manually analyzing these regarding leakage using statistical tools. Empirical verification is not only an error-prone and laborious task but does also not guarantee leakage-freeness on other platforms if no leak is found. As an alternative, formal verification tools like Silver [KSM20], Rebecca [Blo+18], Coco [Gig+21; HB21] and maskVerif [Bar+19] have been proposed which aim at proving the absence of leakage by verifying the respective post-synthesis gate-level netlist. However, these verification tools are tailored to ASIC netlists consisting of logic gates with two inputs and one output. Currently, there does not exist a tool that can handle the structure of FPGA netlists, which consist of LUTs with multiple inputs and up to two outputs. The effects of the FPGA synthesis process on the security of the design are also not understood yet. Therefore, in practice, to ensure security properties are preserved during synthesis, optimizations are usually disabled globally, for example, by selecting

the *keep_hierarchy* option [Cnu+16; Mas+23; MRB18; Wei+19]. While this serves the purpose of ensuring that security-critical design partitioning is kept, it also prevents optimizations in parts of the design that are not security-critical.

Our contribution It is still an open question to which extent the FPGA synthesis process introduces leakage into a masked design. Since existing formal verification tools are built for ASIC netlists, no formal tool exists to detect these leaks on FPGAs. We close this gap by providing the following contributions:

- We present a case study to investigate whether FPGA-specific synthesis flows introduce leakage to masked designs. We investigate two masked multiplication gadgets, for which we formally prove their security on RTL level with an existing ASIC verification tool. Using two different FPGA synthesis tools, we show that the produced netlist is insecure due to glitches introduced by optimization measures such as LUT combining or register retiming. We confirm that the leakage is observable in practice using empirical measurements. (Section 3)
- We present Fenix, a tool that can formally verify the security of (any-order) masked FPGA implementations. Fenix operates directly on the FPGA netlist which allows to detect leakage introduced by optimization measures. Instead of turning off optimizations globally by default, which leads to inefficient designs, designers can apply more specific optimizations and check the security of the resulting design using the tool. (Section 4)
- We show the practicality of Fenix by verifying various masked FPGA implementations, including masked multiplication gadgets, a 2nd-order Keccak S-box, and a full Ascon round. We demonstrate the two operation modes, which allow verification with and without considering glitches. If a leak is found, the exact wire and cycle are automatically localized. (Section 5)
- We publish Fenix on GitHub[1].

2. Background and Related Work

In this section, we cover necessary background on masking, formal verification tools for masked ASIC designs, and describe formal verification using the Fourier expansion of Boolean functions in more detail.

2.1. Masking

The power consumption of a cryptographic device depends mostly on the performed operations and the processed data [CRR02; KJJ99]. Using techniques

[1] https://github.com/kevPretterhofer/FENIX

such as DPA [KJJ99], this dependency can be exploited to recover the secret key used in an encryption. The masking countermeasure aims at breaking this dependency by randomizing sensitive values before starting the cryptographic operation [Cha+99; GP99]. Randomization in a dth-order Boolean masking scheme is achieved by splitting each sensitive value s into $d+1$ shares such that $s = s_0 \oplus ... \oplus s_d$, where $\oplus$ denotes the exclusive OR (XOR) operation. $s_0 ... s_{d-1}$ are chosen uniformly at random and statistically independent of s, while $s_d = s \oplus s_0 \oplus ... s_{d-1}$. Consequently, an attacker observing up to d shares cannot infer any information about the sensitive value. In order to mask a cryptographic algorithm, masked descriptions for the linear and non-linear parts need to be found. Masking linear functions is simple since they be computed individually for each share. Masking non-linear functions is more complex because it requires operations on all shares and usually additional insertion of fresh randomness to avoid unintended direct combinations of shares.

2.2. Leakage sources in masked designs

A masked implementation, even though proven theoretically secure on algorithmic level, is not necessarily secure when implemented on an FPGA. Possible reasons for this are transient timing effects such as glitches and transitions [MPG05; MPO05], which unintentionally combine the shares of a sensitive value. Glitches are counteracted by inserting registers, which serve as synchronization points to balance out different signal propagation times caused by physical circumstances like different wire lengths. Goddard et al. [GLE15] analyze a masked PRESENT S-box implementation without synchronization and show that leakage can be observed when implemented on an FPGA. Roy et al. [Roy+15] study an implementation of a first-order masked SIMON without synchronization and show that glitches and reordering of logic operations lead to leakage when implemented on an FPGA. Finally, Li et al. [Li+20] suggests a simulation-based tool to automatically identify locations where synchronization registers need to be inserted in an FPGA design. Currently, there exists no work which shows that even after inserting synchronization registers, the optimizations performed by EDA tools may again lead to glitches, which in turn cause unintentional share combinations.

2.3. Formal verification of masking

In order to prove that a masked circuit is algorithmically secure and secure in the presence of glitches, formal verification tools consider a specific attacker model. The probing model introduced by Ishai et al. [ISW03] involves an attacker with d probing needles, which can placed on arbitrary wires/gates in a masked circuit. A probe allows capturing the value from a wire for an infinite amount of clock cycles. A masked hardware implementation is dth-order secure in the probing model if the attacker cannot infer any information about the sensitive value by combining the d probed values. The robust probing model [Fau+18] was

introduced as an extension to the original probing model, which allows a probe to observe temporary wire values caused by glitches and transitions.

Several automated tools to check the security of a masked circuit have been proposed. Rebecca [Blo+18] and Coco [Gig+21; HB21] apply Fourier-based techniques to approximate correlation sets (leakage sets) for every gate in the circuit. The leakage set contains all values that can be probed by the attacker by observing the gate output. Other tools like maskVerif [Bar+19], Silver [KSM20], and IronMask [Bel+22] compute these correlation sets exactly, achieving higher accuracy at the cost of efficiency. All these tools first synthesize a masked hardware design with a synthesis tool such as Yosys [Wol16] and then perform verification directly on the gate-level ASIC netlist.

2.4. Fourier-based verification

Rebecca formally verifies the security of masked ASIC circuits based on Fourier expansions of Boolean functions [Blo+18; ODo14]. Given that every gate represents a Boolean function, the correlation of the gate with respect to a sensitive value can be determined using the Fourier expansion (Walsh expansion) [BCG13; Blo+18], which represents the function as a multilinear polynomial. Given the input variables $X = (x_1, x_2, ...x_n)$ with $x_i \in \{-1, 1\}$, the Fourier expansion of the function $f : \{-1, 1\}^n \to \{-1, 1\}$ with $\{-1, 1\} = \{\top, \bot\}$, is defined as:

$$f(X) = \sum_{T \subseteq X} \widehat{f}(T) \prod_{x_i \in T} x_i \tag{1}$$

The real number $\widehat{f}(T)$ is called the Fourier coefficient of f on T, which directly describes the statistical dependence (correlation) of a Boolean function with regard to its inputs. That is, it does not correlate with $T \subseteq X$ iff $\forall T' \subseteq T$ it holds that $\widehat{f}(T') = 0$ [XM88]. In a masked circuit, the inputs of a gate are either shares of a sensitive variable, a fresh random value or any unrelated value such as a control signal. To verify that no 1st-order leakage is exhibited by a gate, we must ensure that the output does not directly correlate with a linear combination of shares, i.e., without fresh randomness. This can be done by computing the Fourier coefficients, and checking that the coefficients of linear share combinations are zero. For dth-order security, Rebecca checks the nonlinear combination of any tuple of d gates.

From a verification perspective it is sufficient to know whether a Fourier coefficient is non-zero or not, while computing the exact value is unnecessary. Therefore, Rebecca groups together all linear combinations of inputs a gate correlates to into correlation sets. A correlation set $\mathcal{C}$ of a gate g computing a function f satisfies the following condition:

$$\text{For } T \subseteq X : \prod_{x_i \in T} x_i \in \mathcal{C}(g) \text{ if } \widehat{f}(T) \neq 0 \tag{2}$$

Correlation sets are sound but incomplete approximations, i.e., linear combinations with zero Fourier coefficients might end up in the correlation set. In order to determine the correlation set of a gate, Rebecca applies propagation rules, starting with the correlation sets of circuit inputs which consist only of the respective input variable. For every gate $g = a * b$, $\mathcal{C}(g)$ is computed by applying the propagation rule for the operator *, considering the correlation sets $\mathcal{C}(a)$ and $\mathcal{C}(b)$.

For example, considering a simple circuit with inputs $X = (p, q, r)$ computing $g(p, q, r) = (p \oplus q) \wedge r$. The Fourier expansion of g is $0.5 + 0.5pq + 0.5r + 0.5pqr$. The correlation set of the first gate $g_1(p, q, r) = p \oplus q$ is $\mathcal{C}(g_1) = \{\{pq\}\}$. The correlation set of g, computed by propagating $\mathcal{C}(g_1)$ and $\mathcal{C}(r)$, is $\mathcal{C}(g) = \{\{\}, \{p, q\}, \{r\}, \{p, q, r\}\}$.

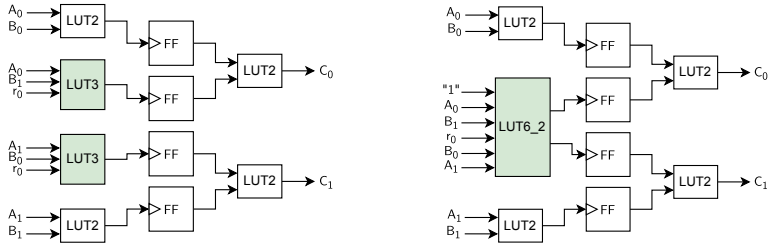
3. Leakage Sources on FPGAs

In this section, we present a case study of leakages introduced to masked hardware implementations when synthesized to an FPGA. We show that two masked multiplication gadgets, which are secure in the presence of glitches, become insecure when implemented on an FPGA. We identify glitches introduced by LUT combining and register retiming, optimization measures applied by the FPGA synthesis process, as the main reason for the observed leakage. We give empirical evidence using physical measurements that the leakage is observable in practice. In the following, we first describe our evaluation setup (Section 3.1), followed by the experiments in which we observe leakage due to LUT combining (Section 3.2) and register retiming (Section 3.3). A LUT with n inputs will be denoted as LUTn.

3.1. Experiment setup

In our case study, we investigate the security of masked multiplication gadgets. Given two shared field elements in $GF(2^n)$, A and B, a masked multiplication gadget computes $C = A \wedge B$ in a secure way. Many masked multiplication gadgets have been introduced in literature [GMK16; ISW03; NRR06; Tri03]. We focus on 1st-order DOM (Domain-Oriented Masking) [GMK16] and 2nd-order ISW (Ishai-Sahai-Wagner) [ISW03].

We use Xilinx Vivado 2022.2 and Xilinx ISE 14.7 to process our designs. Both tools first synthesize the RTL (register-transfer level) design into a netlist consisting of registers, multiplexers and LUTs, and then perform place-and-route, where they assign the netlist components to concrete locations on the FPGA and establish the wiring. Every LUT is physically implemented as a six-input LUT with two outputs (LUT6_2) [Xil16a]. For example, if the post-synthesis netlist contains a LUT4, it is mapped to a LUT6_2 during place-and-route, leaving the unused inputs unconnected. The post-place-and-route netlist is eventually used



(a) Post-synthesis (without LUT combining)

(b) Post-place-and-route (with LUT combining)

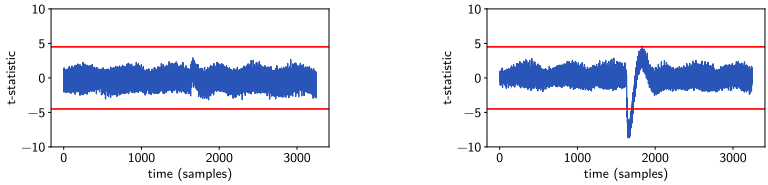
Figure 1.: Netlist of 1st-order DOM multiplier with input shares (A_0, A_1), (B_0, B_1) and fresh randomness r_0, and output shares (C_0, C_1). LUT combining is performed during place-and-route.

to generate the FPGA configuration file (bitstream file).

For the empirical measurements, we use the NewAE CW305 Artix-7 FPGA evaluation board, connected to a PicoScope 6404C at 1.25 Gs/s sampling rate (800 ps sampling interval). The hardware designs operate at a clock frequency of 1.5625 MHz, a submultiple of the sampling rate. We synchronize the clocks between the FPGA and the oscilloscope to reduce the noise level. We apply Welch's t-test following the guidelines of Goodwill et al. [Goo+11] to investigate whether 1st-order leakage is present. For this purpose, a random and fixed set of traces is created. The random set is constructed by assigning fresh random values to the shares of A and B for every trace. For the fixed set, both A and B are set to zero, and fresh values are generated for the shares for every trace. The null hypothesis is that both trace sets have equal means, which can be rejected with a confidence greater than 99.999% if the t-score exceeds -4.5 and 4.5. The RNG is enabled for our measurements.

3.2. LUT combining

LUT combining is an optimization measure that merges several smaller LUTs into a single bigger LUT to save area and reduce the length of the critical path [Xil22a]. LUT combining is either applied during synthesis or during place-and-route. LUT combining during place-and-route means that the post-synthesis netlist contains multiple individual LUTs that get merged when mapped to physical components on the FPGA. In the following case study, we show that for masked hardware designs, LUT combining may merge functions, which are supposed to be computed individually, into a single LUT.



(a) Without LUT combining (b) With LUT combining

Figure 2.: Univariate fixed-vs.-random t-test results for the 1st-order DOM multiplier using 6 million traces. No leakage is visible without LUT combining (a), but the design leaks with LUT combining enabled (b).

LUT combining during place-and-route In our first experiment, we investigate a 1st-order masked DOM multiplication gadget [GMK16]. Given the input sharings (A_0, A_1) and (B_0, B_1) and the fresh random variable r_0, the multiplier computes the output sharing (C_0, C_1) as follows, where registers are indicated by parenthesis:

$$C_0 = (A_0 \times B_0) \oplus \big((A_0 \times B_1) \oplus r_0\big) \tag{3}$$

$$C_1 = \big((A_1 \times B_0) \oplus r_0\big) \oplus (A_1 \times B_1) \tag{4}$$

When synthesized to an ASIC netlist with Yosys, as sketched in Appendix A, the design is 1st-order probing secure, i.e., secure also in the presence of glitches, as we confirm by verification with Coco [Gig+21; HB21].

We synthesize the design with Vivado 2022.2 for an arbitrary chosen 7-series FPGA (xa7s25csga225-2I) using the default settings for synthesis and place-and-route. Figure 1a shows the resulting FPGA netlist of the design after synthesis. The partial multiplication terms $\big((A_0 \times B_1) \oplus r_0\big)$ and $\big((A_1 \times B_0) \oplus r_0\big)$ have been realized using two LUT3s each. The place-and-route process then merges the two LUT3 into a single LUT6_2, as shown in Figure 1b. The first output of the LUT6_2 refers to the first partial multiplication term, and the second output of the LUT6_2 refers to the second partial multiplication term. Internally, a LUT6_2 realizes the two functions using two LUT5s and then uses a multiplexer to realize one output while the other output is directly tied to one of the LUT5s [36], as sketched in Appendix B. Each of the two LUT5s however gets as an input *both* shares of A (A_0, A_1) and both shares of B (B_0, B_1). The functional configurations of the LUT5s are such that the partial multiplication terms are computed, and the unused inputs do not influence the function result once all inputs have stabilized. Still, before all inputs have stabilized, the LUT outputs allow to probe a combination of (in the worst case) all inputs temporarily. The exact combinations that can be observed depend on several things, including the arrival time and ordering of inputs.

Using an empirical evaluation over 6 million traces, we confirm that this leakage can be seen in practical measurements, as shown in Figure 2. Figure 2a presents the t-test results when LUT combining during place-and-route is disabled. As expected, the t-test reveals no significant peaks, which indicates the absence of 1st-order leakage. Figure 2b presents the t-test results when LUT combining is enabled, i.e., following the structure of Figure 1b, which shows significant peaks over the 4.5 border, and therefore, indicates 1st-order leakage. Our evaluation spans four clock cycles, capturing the leaking LUT6_2 computation in the third cycle.

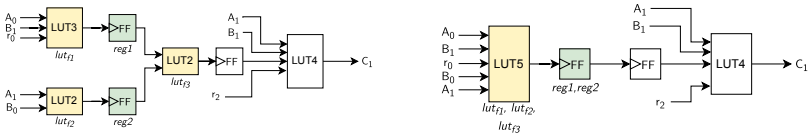
The leak introduced by LUT combining during place-and-route can hardly be detected manually by the designer on RTL level or by looking at the post-synthesis netlist. Manual detection becomes infeasible for larger designs due to the growing complexity, and a possible solution is to disable LUT combining globally. With a formal tool capable of verification on FPGA netlist level, it would be possible to identify single LUTs for which LUT combining must be turned off instead.

LUT combining during synthesis Designs containing many multiplexers, e.g., a multiplexer tree, are realized by combining them into LUTs during synthesis, which potentially causes additional leakage in masked designs. For example, Shahverdi et al. [STE15] introduce an area-optimized TI-Simon implementation using three shares, which splits the round function into three identical component functions. The component function is instantiated exactly once, and another set of shares is sent to the input in each cycle. In order to determine which share is being processed, a single LUT is used that has the shares and the select signals as an input. When the multiplexer select signals glitch, a combination of multiple shares might be visible on the LUT output, resulting in leakage. In an ASIC implementation, this leakage is more challenging to observe because the multiplexers are still individual physical components that take at most one share as an input, and there are many other influencing factors, such as the concrete wiring. By contrast, on an FPGA, a glitch on one of the input select signals might directly allow probing a combination of all shares on the LUT output.

3.3. Register retiming

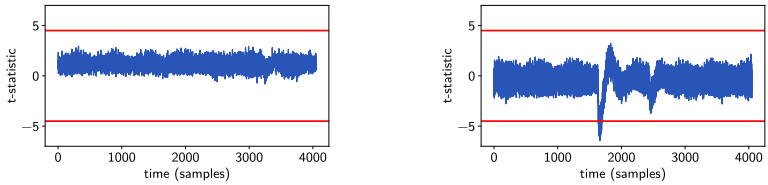
Register retiming, also known as register balancing, allows improving the delay on the critical path of a design by moving the location of registers across combinatorial logic [Dud23; Xil16b; Xil22a]. The original input/output behavior and latency in terms of cycle count remain unchanged, but the possible clock frequency of the resulting design is increased. Retiming is enabled per default when performing synthesis with ISE 14.7.

In masked designs, the insertion and correct location of registers is crucial for security against glitches. Retiming changes the location of registers and, therefore, introduces glitches into the design. In our case study, we investigate a 2nd-order



(a) Post-synthesis without retiming (b) Post-synthesis with retiming

Figure 3.: Netlist of 2nd-order ISW multiplier with input shares (A_0, A_1, A_2), (B_0, B_1, B_2) and fresh randomness r_0, r_1, r_2, and output shares (C_0, C_1, C_2). Only the part of the circuit computing one output share C_1 is shown.



(a) Without register retiming (b) With register retiming

Figure 4.: Univariate fixed-vs.-random t-test results for the 2nd-order ISW multiplier using 30 million traces. No leakage is visible without register retiming (a), but the design leaks with retiming enabled (b).

masked ISW multiplier [ISW03] in which retiming changes the location of registers and share combination happens due to glitches. Again, when synthesized to an ASIC netlist with Yosys, as sketched in Appendix C, we confirm the 2nd-order probing security with Coco. The multiplier computes $C = A \wedge B$ using the random variables r_0, r_1, r_2, resulting in the output sharing (C_0, C_1, C_2), where registers are indicated by parenthesis:

$$C_0 = (A_0 \wedge B_0) \oplus r_0 \oplus r_1 \tag{5}$$

$$C_1 = (A_1 \wedge B_1)\big((r_0 \oplus (A_0 \wedge B_1)) \oplus (A_1 \wedge B_0)\big) \oplus r_2 \tag{6}$$

$$C_2 = (A_2 \wedge B_2) \oplus \big((r_1 \oplus A_0 \wedge B_2) \oplus A_2 \wedge B_0\big) \tag{7}$$

$$\oplus \big((r_2 \oplus A_1 \wedge B_2) \oplus A_2 \wedge B_1)\big) \tag{8}$$

As shown in Figure 3a, when computing C_1 two register stages are required. The intermediate result $(r_0 \oplus A_0 \wedge B_1)$ is computed by a LUT3 (lut_{f1}), while $(A_1 \wedge B_0)$ is computed by a LUT2 (lut_{f2}).

The result of lut_{f1} needs to be stored to a register *reg1* to ensure the refreshing with r_0 is finished before combining it with the result of lut_{f2}. Figure 3b shows the netlist of the circuit after synthesis with ISE 14.7 where register retiming is

applied. The registers *reg1* and *reg2* are moved forward, i.e., they are swapped with lut_{f3}. Then, the synthesizer merges lut_{f1}, lut_{f2} and lut_{f3} into a single LUT5, and registers *reg1* and *reg2* can be merged. The area of the design is reduced, since the resulting circuit only requires two LUTs instead of four, and two registers instead of three. However, *reg1* ensuring proper refreshing is removed and consequently, an attacker probing the output of the LUT5 can learn a function of two shares of A and B respectively due to glitches, for example, if the randomness r_0 arrives later at the LUT5 input. Note that in this example, LUT combining happens besides register retiming, but the leakage would also be present if LUT combining was disabled.

Intuitively, the computations of C_0, C_1, and C_2 individually must be 1st-order secure such that the whole gadget can be 2nd-order secure. Therefore, in the empirical measurements, we assess the 1st-order security of the part of the circuit computing C_1. Figure 4 shows the results of our empirical leakage evaluation using 30 million traces, which confirms that register retiming introduces 1st-order leakage to the design. More specifically, we compare the security of the design without register retiming (Figure 4a) to the security of the design when retiming is enabled (Figure 4b). In the latter case, we perceive peaks in the t-score over the 4.5 border as expected, indicating 1st-order leakage.

The leak introduced by register retiming was not detected by Coco because the ASIC netlist did not contain the retimed registers. To spot the leakage, the post-synthesis FPGA netlist needs to be considered.

4. Fenix

In this section, we describe how we built Fenix, the first tool for the formal verification of masked FPGA implementations. Fenix operates directly on post-place-and-route netlist level and therefore easily detects leaks introduced by synthesis and place-and-route. Similar to Rebecca/Coco, our tool utilizes Fourier expansions and approximates correlation sets for LUTs, registers, and multiplexers, which are part of the FPGA netlist. The correlation sets are encoded and then checked with a SAT solver. First, we describe the high-level verification flow of Fenix, give a precise description of the input netlist produced by ①, and describe the verification step (④) in detail.

4.1. Verification Flow

The verification flow implemented by Fenix consists of six steps, as shown in Figure 5, divided into three preprocessing steps ①-③, the verification step ④, the solving step ⑤, and the interpretation step ⑥:

① The masked hardware design written in an HDL such as Verilog or VHDL is processed by the FPGA design flow, which typically consists of synthesis and place-and-route. In our experiments, we focus on Xilinx tools, although the

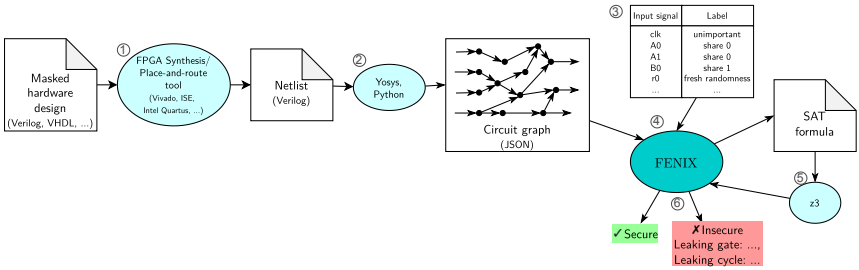


Figure 5.: Verification flow of Fenix, consisting of ①-③ three preprocessing steps, ④ the actual verification step, ⑤ the SAT solving step and ⑥ the interpretation step.

verification flow could, in principle, be used with tools from other vendors such as Intel or Lattice as well. The result of this process is the post-place-and-route netlist in Verilog format. LUTs that are combined into a LUT6_2 are marked as pairs using dedicated annotations.

② Using Yosys [Wol16], we parse the netlist and transform it into a circuit graph with gates as nodes and wires as edges. The graph is stored in JSON format. Using a dedicated preprocessing script, we merge the LUTs, which are marked as pairs in the netlist, into a single LUT6_2 element. Additionally, we remove cycles in the circuit by unrolling and then topologically sort the graph such that the root nodes are registers and circuit inputs.

③ A label is assigned to every circuit input bit, which expresses its purpose in the masking scheme. An input bit can either be a *share*, *random* or *unimportant*.

④ For each node in the circuit graph, Fenix computes the respective correlation set according to the propagation rules and encodes them into a SAT formula with regard to the masking order. For LUTs, Fenix determines the propagation rules in a precomputation step. This process is repeated for a specific number of clock cycles for a specific masking order chosen by the user. Similar to Rebecca/Coco, Fenix supports both stable and transient verification.

⑤, ⑥ The resulting SAT formula is given to the Z3 Theorem Prover, which searches for leaks in the correlation sets over all cycles. If a leak is found, the SAT formula is satisfiable, and a possible variable assignment is given to the Fenix tool. The tool interprets the model and reports the leaking gate and cycle. If no leak is found, the SAT formula is unsatisfiable, and the tool reports that the circuit is secure.

4.2. Input netlist

Fenix operates on the post-place-and-route netlist in Verilog format. In Vivado, it can be extracted using the `write_verilog` TCL command. The netlist describes

the design as a graph where nodes are either circuit inputs, outputs, LUTs, registers, or multiplexer cells, and edges are wires. For every LUT, the netlist contains the initialization (INIT) string, which represents the output values of the LUT for every possible assignment of the inputs and eventually allows to determine the logic function.

On the FPGA, LUTs are grouped into slices, and slices are grouped into CLBs (Configurable Logic Blocks), which are connected to a switch matrix for routing [Xil16a]. Slices are categorized into SLICEL and SLICEM. SLICEL consist of four LUT6_2, eight registers, several cascade multiplexers, and logic elements to realize carry logic efficiently. SLICEM additionally support storing data using distributed RAM and contain shift registers. Which features are used depends on the configuration of the slice. For most cryptographic implementations, SLICEL are configured to process an input using the LUTs and storing either the result to the register or forwarding it to another LUT. In this case, the post-place-and-route netlist verified by Fenix will contain only the LUTs and registers if used because the post-place-and-route netlist replicates the exact configuration of slices. The cascade multiplexers and carry logic are not included in the netlist, and therefore not considered by the verification, even though they are physically present. However, it is valid to exclude these logic elements from the verification because they are inactive, i.e., do not actively compute anything, and can therefore not cause any share-dependent combinations leading to leakage in the design. For example, as sketched in Appendix D, the three cascade multiplexers each connect two LUT6_2 outputs, and the third multiplexer connects the output of the two multiplexers before. If the multiplexer select signals glitch, the attacker could, in the worst case, probe a combination of the output of all four LUTs in the slice. However, in practice, the select signals are stable and never change their value, which allows to probe only the output of a single LUT and therefore does not give the attacker any advantage. In case any implementation ever requires the use of the cascade multiplexers and carry logic, which is very rare for cryptographic implementations, the slice configuration is changed, and the respective elements are added to the netlist.

4.3. Verification of LUTs

Before starting verification with Fenix, the user needs to provide a label file indicating the purpose of each input signal. We incorporate the labeling system of Rebecca/Coco, that is, a *share* represents a share of a sensitive bit, *random* means a fresh and uniformly-distributed random bit, and *unimportant* means that it is not relevant for the masked implementation, e.g., a control signal or the clock input. These labels are directly translated into correlation sets for the input signals of the circuits. Then, Fenix propagates these correlation sets through the circuit according to the propagation rules. Which propagation rule is used depends on the gate type (input, register, LUT) and the verification mode (stable or transient). The stable verification mode refers to checking the security

Table 1.: Overview of propagation rules

Gate type	Function	Propagation rule	
		Stable $\mathcal{S}^t(g)$	**Transient** $\mathcal{T}^t(g)$
Input	$x \in X$	$\{x\}$	$\{x\}$
Register	Copy	$\mathcal{S}^{t-1}(g)$	$\mathcal{S}^{t-1}(g)$
Multiplexer	$g = g_3?g_1 : g_2$	$\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_2) \cup (\mathcal{S}^t(g_3) \triangle \mathcal{S}^t(g_1)) \cup (\mathcal{S}^t(g_3) \triangle \mathcal{S}^t(g_2))$	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 3$
LUTn	$g = f(g_1, ...g_n)$	Derived from INIT	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq n$
LUT6_2	$g = f_1(A)$, $A \subseteq (g_1, ...g_5)$	Derived from INIT_g	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 5$
	$h = f_2(B)$, $B \subseteq (g_1, ...g_5)$	Derived from INIT_h	$\triangle_i(\emptyset \cup \mathcal{T}^t(g_i))$ for $1 \leq i \leq 5$

in the classic probing model [ISW03], while the transient verification mode allows glitch-extended probes. In this work, we propose propagation rules for LUTs for both stable and transient verification for the first time . The propagation rules for registers and multiplexers are carried over from Rebecca/Coco.

Like Coco, Fenix generally performs verification in the time-constrained probing model. The time-constrained probing model [Gig+21] restricts the temporal scope of a probe to one clock cycle, i.e., a probe is used to observe information in one specific clock cycle at one specific location. By that, Fenix can be used to verify non-pipelined circuits, although it is restricted to verifying circuits without state machines and control signals. Therefore, we formulate propagation rules (correlation sets) implemented by Fenix in a cycle-aware fashion, which allows a more intuitive interpretation of results and several optimizations. In Table 1 we give an overview of the propagation rules used by Fenix. $\triangle$ refers to the symmetric set difference.

Propagation rules for LUTn In the stable case, the correlation set computed according to the propagation rule reflects the correlation at the LUT output, assuming no glitches on the input. Before verification starts, we determine a stable propagation rule for each LUT in the netlist by computing the Fourier representation of the LUT's function. In order to determine these propagation rules for a LUT representing the function $g = f(g_1, ...g_n)$, we apply a three-step process inspired by [ODo14]:

1. Every LUTn in the circuit graph is associated to a hexadecimal initialization string which represents the output values of the LUT for every possible assignment of inputs in ascending order. We convert the initialization string to a 2^n-digit binary number $\mathsf{INIT} = \{-1, 1\}^n$, and then compute the truth table Y:

$$\forall a \in \{-1, 1\}^n : Y[a] = \mathsf{INIT}[pos(a)] \tag{9}$$

The function $pos(a)$ converts the n-bit binary vector a into the index such that $f(a) = \mathsf{INIT}[pos(a)]$.

2. We compute the indicator polynomials $\mathbb{1}_a$. Intuitively, $\mathbb{1}_a(g_1, ...g_n) = 1$ if $a = (g_1, ...g_n)$, else it is 0:

$$\forall a \in \{-1, 1\}^n : \mathbb{1}_a(g_1, ...g_n) = \prod_{i=1}^{n} \frac{1 + a_i g_i}{2} \tag{10}$$

3. Using the indicator polynomials we arrive at the Fourier representation of f by computing and summarizing the following expression:

$$f(g_1, ..g_n) = \sum_{a \in \{-1,1\}^n} Y[a] \cdot \mathbb{1}_a(g_1, ...g_n) \tag{11}$$

From that expression, we extract the linear combinations of the inputs and the Fourier coefficients, which enables the construction of the correlation set $\mathcal{S}^t(g)$ according to Equation 2. The propagation rule is then formed by replacing the abstract LUT inputs by the respective input correlation sets $\mathcal{S}^t(g_1), ...\mathcal{S}^t(g_n)$.

In the transient case, FENIX makes the worst case assumption and extends the attacker's abilities by assuming any arbitrary Boolean function from the LUT's original inputs may be probed, independent of the INIT string or the concrete ordering of inputs. This is reflected by the respective propagation rule as shown in Table 1.

Example Consider a LUT3 with initialization string 0x78 = $(01111000)_2$, representing the function $f(g_1, g_2, g_3) = (g_1 \wedge g_2) \oplus g_3$. To compute the propagation rule for the stable mode we first convert 0x78 to INIT = (1,-1,-1,-1,1,1,1). By computing the truth table and indicator polynomials we arrive at the Fourier expansion $f(g_1, g_2, g_3) = 0.5g_3 + 0.5g_1g_3 + 0.5g_2g_3 - 0.5g_1g_2g_3$, and the correlation set with regard to abstract inputs $\mathcal{S}(g) = \{\{g_3\}, \{g_1, g_3\}, \{g_2, g_3\}, \{g_1, g_2, g_3\}\}$. This yields the propagation rule $\mathcal{S}^t(g) = \mathcal{S}^t(g_3) \triangle (\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_3)) \triangle (\mathcal{S}^t(g_2) \cup \mathcal{S}^t(g_3)) \triangle (\mathcal{S}^t(g_1) \cup \mathcal{S}^t(g_2) \cup \mathcal{S}^t(g_3))$. In the transient case, the propagation rule says to propagate all possible combinations of the transient correlation sets, i.e., $\mathcal{T}^t(g) = (\{\emptyset\} \cup \mathcal{T}^t(g_1)) \triangle (\{\emptyset\} \cup \mathcal{T}^t(g_2)) \triangle (\{\emptyset\} \cup \mathcal{T}^t(g_3))$.

Propagation rules for LUT6_2 A LUT6_2 consists of two LUTs connected by a multiplexer (cf. Appendix B), that have the common inputs $(g_1, ...g_5)$. Each LUT operates on a subset of inputs, i.e., the first LUT computes $g = f_1(A), A \subseteq (g_1, ...g_5)$ and the second LUT computes $h = f_2(B), B \subseteq (g_1, ...g_5)$. The sixth input is driven high, which ensures that the output of the first LUT is forwarded to the first output port, and the output of the second LUT is forwarded to the second output port. In the stable case, we compute the propagation rules for

Table 2.: Verification of masked implementations using Fenix. ✖ indicates intentionally broken implementations. Nodes include inputs, outputs, registers, multiplexers and LUTs.

Name	Nodes	Cycles	Input shares	Fresh randomness	Runtime Stable	Runtime Transient
1st-order						
DOM-AND [GMK16]	107	2	4 × 8 bit	8 bit	0.1 s	0.1 s
DOM-AND [GMK16] with LUT combining	20	2	4 × 1 bit	1 bit	0.1 s	< 0.1 s ✖
DOM-AND [GMK16] without registers	74	1	4 × 8 bit	8 bit	<0.1 s	<0.1 s ✖
TI-AND [NRR06]	96	1	6 × 8 bit	-	<0.1 s	<0.1 s
ISW-AND [ISW03]	155	3	4 × 8 bit	8 bit	0.3 s	0.6 s
Trichina-AND without registers [Tri03]	64	1	4 × 8 bit	8 bit	< 0.1 s	0.6 s ✖
DOM Keccak S-box [GSM17]	62	2	10 × 1 bit	5 bit	<0.1 s	0.6 s
DOM Ascon Round	5136	3	10 × 64 bit	320 bit	9 min	9.5 h
DOM AES S-box [GMK16]	409	5	2 × 8 bit	28 bit	4.5 min	-
2nd-order						
DOM-AND [GMK16]	219	2	6 × 8 bit	24 bit	2.1 s	19 s
ISW-AND [ISW03]	315	3	6 × 8 bit	24 bit	42 s	1.8 m
ISW-AND [ISW03] with retiming	29	4	6 × 1 bit	3 bit	0.3 s	< 0.1 s
DOM Keccak S-box [GSM17]	62	2	15 × 1 bit	15 bit	0.3 s	1.7 s
3rd-order						
DOM-AND [GMK16]	49	2	8 × 1 bit	6 bit	<0.1 s	0.1 s
DOM-AND [GMK16]	371	2	8 × 8 bit	48 bit	1.3 min	1.9 h

the two individual LUTs, and use the rules to assign the correlation sets to the respective output. Conceptually, we treat the two LUT5s independently from each other. In the transient case, we however need to take into account that all five inputs are forwarded to each of the two LUTs, and an attacker might probe an arbitrary combination of all five inputs even though not all inputs are processed functionally. Therefore, we assign the same correlation set to both outputs to capture the fact that all inputs $(g_1, ...g_5)$ enter both LUTs. Similar to the regular LUTn, we assume the attacker can probe an arbitrary function of input signals.

5. Evaluation

In this section, we first describe the evaluation setup and then discuss and interpret the results obtained from verifying different masked implementations. We compare our tool to others, including Rebecca [Blo+18], Coco [Gig+21], maskVerif [Bar+19], Silver [KSM20], and discuss possible optimizations to improve the tool for the future.

5.1. Setup

For evaluating Fenix in terms of verification runtime, we use a notebook with an AMD Ryzen 5 4500U CPU with 16 GB of RAM. To synthesize the designs, we use Vivado 2021.1 and a Xilinx 7 Series FPGA as the target device (XC7S75-FGGA676-1). For our experiments, we use the default synthesis options of Vivado with a few exceptions. First, we set the Verilog attribute `extract_reset` to the reset signals in our designs. This ensures that the reset signal is directly connected to the respective register instead of being used as an input to the LUT before the register. Unless stated otherwise, we set the options `-no_lc` to true to prevent LUT combining (cf. Section 3.2), and `-retiming` to false to prevent register retiming (cf. Section 3.3) to construct secure FPGA designs. Instead, we could also have placed `dont_touch` attributes on selected wires.

5.2. Results

The verification results of the masked hardware implementations and the verification runtime are shown in Table 2. We evaluate several 1st-, 2nd- and 3rd-order masked hardware implementations. Table 2 shows the name of the masked design, the number of nodes (inputs, outputs, registers, multiplexers, LUTs) in the circuit graph, the number of verified cycles, and the number of labels provided by the user (shares and fresh randomness). In terms of runtime, we report the total verification runtime for the stable and transient mode, which includes the steps ④-⑥.

The selection of masked circuits covers various 1st-order masked AND gadgets [GMK16; ISW03; NRR06; Tri03] which can all be verified in less than a second. We include a DOM-AND implementation without a register stage, which is secure in the stable case but exhibits leakage in the transient case due to glitches, which is correctly detected by Fenix. During our analysis, we make an interesting observation about the security of the Trichina-AND gadgets, which we implement without a register stage. In the transient case, the implementation is therefore insecure due to glitches, as confirmed by our tool. However, the Trichina-AND gadget passes the stable verification on the FPGA netlist with Fenix, but fails stable verification with Rebecca when implemented on an ASIC as shown in [Blo+18]. The reason for this is that in the case of Rebecca, the ASIC synthesis tool reorders the individual binary gates, but the specific order of gates is crucial for stable security. By contrast, when implemented on an FPGA, the binary gates are merged into a single LUT, leaving less possibilities for reordering.

Fenix can also be applied to bigger designs, like the Keccak S-box. We successfully verify a complete Ascon round consisting of more than 5000 nodes using 64-bit input shares in less than 10 hours in the transient mode. By verifying 2nd- and 3rd-order DOM-AND gadgets, and a 2nd-order ISW-AND gadget, we show that Fenix can be applied to higher orders. We successfully verify a 1st-order masked AES S-box protected by DOM with a latency of five cycles

in 5 minutes for the stable case. In the transient case, the solver returned no result after one week, although Fenix managed to build the formula in 3 seconds. This is mainly due to the complex structure of an AES S-box, and to choosing Z3 as a solver. Also, in the original publication of Rebecca [Blo+18], the authors mention that they checked each sensitive bit separately while treating the others as constants and starting the verification eight times instead of labeling all sensitive bits at once, which significantly reduces the complexity of the SAT problem. The reduction of the complexity can, for example, be seen very well when comparing the verification runtimes of Fenix when verifying a 3rd-order DOM-AND gadget. For 1 bit, the verification finishes in 0.1 s, while for 8 bits, it takes 1.9 h.

5.3. Comparison and Future Work

Currently, no verification tool operating on FPGA netlists exists, which does not allow for a direct and fair comparison with another state-of-the-art tool. Also, comparisons with other ASIC verification tools need to be made with caution, as these tools often use parallelization instead of only one CPU core. Compared to Rebecca, Fenix provides similar and, in most cases, even better verification runtimes, given the fact that Rebecca uses multithreading and focuses on 1-bit implementations. Coco, which was built on top of Rebecca for verifying masked software implementations on CPU netlists, was recently extended for hardware implementations [HB21]. Coco is able to provide very low verification runtimes due to extensive simplification of correlation sets based on the behavior of ASIC gates. For example, it tracks whether the input of an AND gate is zero and glitch-free, allowing to safely assume that the output will also be zero and glitch-free. Such assumptions are not possible in the case of LUTs since the exact internal structure of LUTs, and therefore, their glitch behavior, is unknown. Additionally, Coco considers control signals by reading simulation traces of the respective designs, which also helps to simplify stable correlation sets. A similar technique could be integrated into Fenix' stable verification mode and would require to re-compute the propagation rules for LUTs in every cycle depending on the value of a concrete control signal.

The verification runtimes of basic gadgets are also comparable to the ones reported by maskVerif and Silver. For future work, it would be very interesting to check other properties such as composability, e.g., strong non-interference (SNI) [Bar+16] with Fenix as it is already done by maskVerif and Silver.

The total verification runtime of Fenix is largely determined by the time it takes to solve the generated SAT formula. For example, for verifying the ASCON Round, it took only 15 s to create the SAT formula (step ④), while it took 9 min to solve it. The evaluation of different solvers is therefore also an open point for future work, as we believe a solver better suited for our problem would further decrease the verification runtimes drastically.

6. Conclusion

In this paper, we investigated the effect of the FPGA synthesis process on the security of masked hardware designs. We showed that optimization measures such as LUT combining and register retiming could introduce additional leakage into a design that was formally verified to be secure even in the presence of glitches. Using empirical measurements, we demonstrated that the leakage is also observable in practice. These leaks can hardly be detected manually by the designer on RTL level and usually require a close inspection of the post-synthesis or post-place-and-route netlist, which becomes infeasible for larger designs due to the growing complexity. Based on this case study, we therefore presented Fenix, the first formal verification tool to verify masked FPGA designs directly on netlist level. More concretely, we show how the Fourier analysis of Boolean functions can be used to determine propagation rules for LUTs in a circuit, which can then be used to estimate correlation sets during the verification. We evaluated the tool using several masked designs, including multiplication gadgets and a full Ascon round.

Acknowledgements

This research is supported by the Austrian Science Fund (FWF SFB project SPyCoDe F8504). We thank the anonymous reviewers for their valuable feedback.

Appendix A. 1st-order DOM multiplier

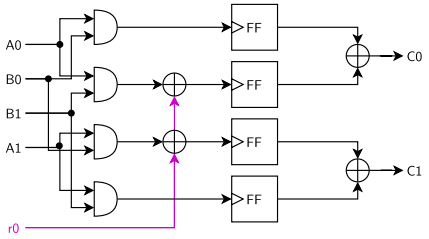


Figure 6.: 1st-order DOM multiplication gadget [GMK16] represented as an ASIC netlist.

Appendix B. Internal structure of LUT6_2 primitive

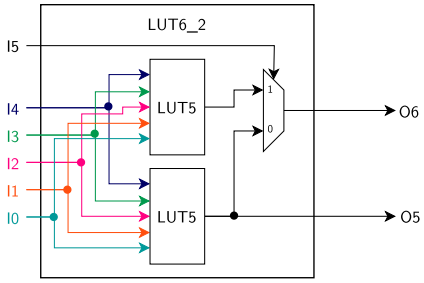


Figure 7.: Internal structure of a LUT6_2, which consists of two LUT5 and a multiplexer [Xil22b].

Appendix C. 2nd-order ISW multiplier

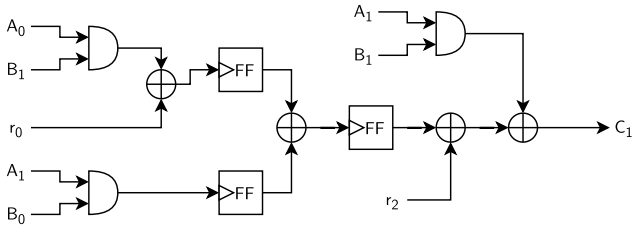


Figure 8.: 2nd-order ISW multiplication gadget [ISW03] represented as an ASIC netlist.

Appendix D. SLICEL with cascade multiplexers

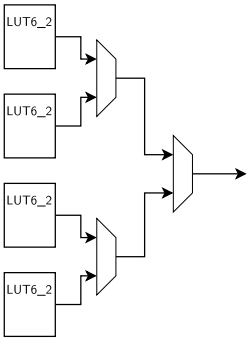


Figure 9.: Simplified sketch of SLICEL connecting four LUT6_2s and cascade multiplexers.

References

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 116–129.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[BCG13] Shivam Bhasin, Claude Carlet, and Sylvain Guilley. "Theory of masking with codewords in hardware: low-weight dth-order correlation-immune Boolean functions". In: *IACR Cryptol. ePrint Arch.* (2013), p. 303.

[Bel+22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. "IronMask: Versatile Verification of Masking Security". In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 142–160.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Cas+21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. "Hardware Private Circuits: From Trivial Composition to Full Verification". In: *IEEE Trans. Computers* 70.10 (2021), pp. 1677–1690.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cnu+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Masking AES with d+1 Shares in Hardware". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 194–212.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference". In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555. DOI: 10.1109/TIFS.2020.2971153. URL: https://doi.org/10.1109/TIFS.2020.2971153.

[Dud23] Chaithanya Dudha. *Retiming in Vivado Synthesis*. https://support.xilinx.com/s/article/934201?language=en_US. Retrieved on 23/05/2023. 2023. URL: https://support.xilinx.com/s/article/934201?language=en_US.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing

Model". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GLE15] Zachary N. Goddard, Nicholas LaJeunesse, and Thomas Eisenbarth. "Power analysis of the t-private logic style for FPGAs". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE Computer Society, 2015, pp. 68–71.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A testing methodology for side-channel resistance validation". In: *NIST Non-Invasive Attack Testing Workshop*. 2011.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 158–172.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 2017, pp. 205–212.

[Gün11] Tim Güneysu. "FPGAs in Cryptography". In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, pp. 499–501.

[HB21] Vedad Hadzic and Roderick Bloem. "COCOALMA: A Versatile Masking Verifier". In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.34727/2021/isbn.978-3-85448-046-4_9. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4%5C_9.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. “Private Circuits: Securing Hardware against Probing Attacks”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. “SILVER - Statistical Independence and Leakage Verification”. In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[Li+20] Yanbin Li, Ming Tang, Yuguang Li, and Huanguo Zhang. “A pre-silicon logic level security verification flow for higher-order masking schemes against glitches on FPGAs”. In: *Integr.* 70 (2020), pp. 60–69.

[Mas+23] Loïc Masure, Pierrick Méaux, Thorben Moos, and François-Xavier Standaert. “Effective and Efficient Masking with Low Noise Using Small-Mersenne-Prime Ciphers”. In: *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14007. Lecture Notes in Computer Science. Springer, 2023, pp. 596–627.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. “Side-Channel Leakage of Masked CMOS Gates”. In: *Topics in Cryptology - CT-RSA 2005, The Cryptographers’ Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. Springer, 2005, pp. 351–365.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. “Successfully Attacking Masked AES Hardware Implementations”. In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 157–171.

[MRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. "Multiplicative Masking for AES in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 431–468.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[ODo14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5.

[Pra+04] Norbert Pramstaller, Stefan Mangard, Sandra Dominikus, and Johannes Wolkerstorfer. "Efficient AES Implementations on ASICs and FPGAs". In: *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*. Ed. by Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa. Vol. 3373. Lecture Notes in Computer Science. Springer, 2004, pp. 98–112.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[Roy+15] Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Jean-Luc Danger, and Debdeep Mukhopadhyay. "From theory to practice of private circuit: A cautionary note". In: *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*. IEEE Computer Society, 2015, pp. 296–303. DOI: 10.1109/ICCD.2015.7357117. URL: https://doi.org/10.1109/ICCD.2015.7357117.

[SSM21] Rajat Sadhukhan, Sayandeep Saha, and Debdeep Mukhopadhyay. "Shortest Path to Secured Hardware: Domain Oriented Masking with High-Level-Synthesis". In: *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 223–228.

[STE15] Aria Shahverdi, Mostafa Taha, and Thomas Eisenbarth. "Silent Simon: A threshold implementation under 100 slices". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE Computer Society, 2015, pp. 1–6.

[Tri03] Elena Trichina. "Combinational Logic Design for AES SubByte Transformation on Masked Data". In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 236.

[Wei+19] Yongzhuang Wei, Fu Yao, Enes Pasalic, and An Wang. "New second-order threshold implementation of AES". In: *IET Inf. Secur.* 13.2 (2019), pp. 117–124.

[Wol16] Claire Wolf. *Yosys Open Synthesis Suite*. http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021. 2016.

[Xil16a] Xilinx. *7 Series FPGAs Configurable Logic Block User Guide (UG474)*. https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB. Retrieved on 21/05/2023. 2016. URL: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB.

[Xil16b] Xilinx. *Vivado Design Suite User Guide Implementation (UG904)*. https://docs.xilinx.com/v/u/2016.2-English/ug904-vivado-implementation. Retrieved on 23/05/2023. 2016. URL: https://docs.xilinx.com/v/u/2016.2-English/ug904-vivado-implementation.

[Xil22a] Xilinx. *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*. https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology. Retrieved on 23/05/2023. 2022. URL: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology.

[Xil22b] Xilinx. *Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)*. https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/LUT6_2. Retrieved on 23/05/2023. 2022. URL: https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/LUT6_2.

[XM88] Guo-Zhen Xiao and James L. Massey. "A spectral characterization of correlation-immune combining functions". In: *IEEE Trans. Inf. Theory* 34.3 (1988), pp. 569–571.

Part III.

Conclusions and Outlook

'Cause there were pages turned
with the bridges burned
Everything you lose is a step you take

Taylor Swift – You're On Your Own, Kid

9 Conclusions and Outlook

Masking is one of the most popular and most researched countermeasures against power analysis attacks. It follows the idea of decoupling the intermediate values processed by a cryptographic device from the sensitive values, such as the secret key, by splitting the sensitive values into multiple random shares. One very beneficial aspect of masking is that it provides provable security based on assumptions like the *independent leakage assumption* (ILA). The ILA states that the power consumption of independent computations leaks shares independently instead of their combinations. Based on an algorithmic description, a masking scheme is either implemented in hardware or in software. However, physical effects like glitches and transitions that occur in CMOS circuits have been shown to potentially violate the ILA, leading to insecure masking schemes. In this thesis, we investigated the security and efficiency of masked software and hardware implementations, taking into account effects that potentially violate the ILA.

Conclusions in the Context of Masked Software We studied the security of masked software implementations when executed on CPUs and identified microarchitectural components that can cause violations of the ILA. More specifically, we observed that physical effects like glitches and transitions that occur in the CPU on gate-level can lead to the temporary combination of intermediate values and, therefore, cause leakage during the execution of masked software. In [Gig+21], we demonstrated, based on the example of the RISC-V Ibex core, that components like the register file, computation units (including the ALU), and the load-store unit are prone to such effects. Similar observations can be made for more complex processor architectures, as we demonstrated in [GPM21] by analyzing the RISC-V SweRV core, which features a 9-stage pipeline with forwarding logic and superscalar building blocks. We formalized our findings by adapting the order-reduction theorem, such that the number of pipeline stages

and execution units are considered.

To deal with ILA violations originating from components in the CPU, we showed that changes are required both on hardware and software level. In [Gig+21], we proposed a list of small changes for the Ibex core that eliminate most of the leakages and allow the secure execution of masked software if the software fulfills a set of constraints. In [GPM21], we reported similar findings for the SweRV core and discussed how software constraints can be adapted for more complex microarchitectures in general.

In [GPM23b], we further extended this analysis to include the embedded OS that runs the masked software as one out of many tasks. We concluded from this analysis that ILA violations are potentially also committed by the embedded OS, for instance, during context switches. To fix these issues, we explored several strategies that involve changes to the context-switching routine of the OS.

Our analysis clearly highlights that the CPU needs to be taken into consideration when constructing masked software implementations. In this context, we introduced a new formal verification approach that can be used to prove the security of a masked software implementation given a concrete CPU netlist.

Conclusions in the Context of Masked Hardware Masking for hardware implementations does not come free of charge but rather with a significant overhead in latency, randomness, and area. A considerable amount of resources, for instance, RNGs and additional register stages, is spent on dealing with ILA invalidations caused by glitches and transitions. In [Gig+24a] we showed for a second-order masked AES design that the randomness consumption can be significantly reduced by recycling fresh randomness produced by the RNG, and using shares of unrelated state bytes for refreshing in a COTG-based fashion. We demonstrated that the reduction of randomness does not necessarily lead to an increased encryption latency in the design and can be done while keeping the minimal number of three shares for second-order security.

Formal verification is one option to check whether a masked hardware implementation is secure in the presence of glitches and transitions. Most existing formal verification tools focus on Boolean masking, although arithmetic masking has become more important recently, especially with the rise of PQC. In [GPM23a], we showed how existing Boolean verification approaches can be extended to the arithmetic domain. We demonstrated that this extension allows to verify A2B/B2A conversion algorithms and uncovered a possible issue related to glitches when implementing a popular A2B/B2A conversion algorithm in hardware. In a further analysis, we showed that the developed approach can also be applied in the software domain.

In the context of masked hardware implementations on FPGAs, we showed in [GPM24] that optimizations performed during the synthesis process could introduce glitches into the design that lead to invalidations of the ILA. From this analysis, we conclude that even when using a glitch-resistant masking scheme like

DOM, there is no guarantee that the glitch resistance is still given after FPGA synthesis. To identify such issues, we proposed a new formal verification tool that works directly on the post-synthesis FPGA netlist.

9.1. Future work

This thesis enhances the practical security and efficiency of masked software and hardware implementations. However, plenty of interesting open problems and new research directions are still available. In general, we believe it is necessary to further narrow the gap between the theoretical and practical security of masked implementations to counteract side-channel attacks efficiently. The following paragraphs summarize possible next steps towards reaching this goal.

Automated Masking in Software In this thesis, we proposed software constraints for masked Assembly implementations that currently need to be integrated manually. Clearly, this represents a laborious and error-prone task that a compiler could easily automate. Another open problem in this context is that masked implementations constructed in high-level languages such as C usually cannot be trusted because compilers perform optimizations like reordering instructions or changing register allocations, which potentially break security guarantees made for the initial masking scheme. One option would be to make the compiler aware of the masking countermeasure such that only security-uncritical optimizations are performed. Another option would be to generate masked assembly implementations directly from high-level descriptions of unmasked cryptographic primitives by also taking into account possible leakage coming from the CPU hardware. While there already exists some work in that direction, mainly for masked hardware implementations [BSG23; Cas+24; Kni+22; Wu+23b], the state-of-the-art for software is still rather rudimentary, especially considering microarchitectural leakage.

Enhanced Formal Verification Tools Existing verification tools can mainly be used to check small cryptographic building blocks, such as masked S-boxes, instead of larger ciphers over multiple rounds or bigger PQC implementations. In general, the scalability of these tools is limited, and they are less applicable for higher masking orders or a larger number of gates/execution cycles, especially when it comes to complex dependencies between intermediate variables, e.g., when reusing randomness. Scalability can be improved by approximations, which, however, leads to less accuracy. Therefore, finding a good tradeoff between scalability and accuracy is a central aspect of improving formal verification tools. Additionally, existing approaches work with adversary models, which mostly include glitches and transitions but largely ignore coupling and crosstalk effects. As CMOS technology evolves and 2nm processes will be used for manufacturing

semiconductors soon, the distances between wires on the chip will shrink, provoking crosstalk. Therefore, such effects on the security of masked software need to be formalized such that integrating them into formal verification tools is possible.

Broadening of the Verification Scope In this thesis, we assumed that at the beginning of the execution of a masked implementation, shares magically appear in the register file or the state registers. In reality, the cryptographic implementation is often embedded in a SoC, connecting many different components via a shared bus. For example, a typical SoC could contain a cryptographic RNG to generate fresh randomness and the initial sharing and a CPU to execute the masked software but accelerates certain operations with a dedicated cryptographic coprocessor. In this case, the attack surface is broadened by many aspects, including the bus used to transport the shares. Investigating the security of SoCs as a whole, in turn, requires appropriate verification techniques. Considering the high complexity of such systems, there is presumably little hope that empirical verification can deliver meaningful insights, again raising the need for improved formal verification techniques.

Reducing Randomness Consumption We demonstrated that reusing randomness in masked hardware implementations greatly decreases the overhead of a masked implementation in terms of e.g., area. While we focused on AES, the applicability of the concept to other ciphers makes an interesting future research direction. First steps in that direction have already been made, e.g., for KETJE [ANR19], Ascon and Keyak [SD17], ARX ciphers [JPS18], and Prince [MMM21], although focusing instead on first-order hardware implementations. Reusing randomness in higher-order ($d > 2$) designs is still an open problem, mainly because existing concepts were derived manually, and for higher orders, it likely requires a more systematic approach. Related to that, it would also be interesting to try to derive a generic approach that works for any masking order d. Furthermore, existing work almost exclusively focuses on masked hardware implementations, although the runtime of masked software could also be decreased significantly when reducing the required amount of fresh randomness. In this context, the low-noise setting of masked software implementations could be problematic, which becomes even worse with less RNG utilization. This thesis also discusses A2B/B2A conversion algorithms, which require a certain amount of fresh randomness but have not yet been studied in terms of reusing randomness.

Part IV.

References and Appendices

Bibliography

[ACS18] Victor Arribas, Thomas De Cnudde, and Danilo Sijacic. "Glitch-Resistant Masking Schemes as Countermeasure Against Fault Sensitivity Analysis". In: *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2018, Amsterdam, The Netherlands, September 13, 2018*. IEEE Computer Society, 2018, pp. 27–34.

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. "An Implementation of DES and AES, Secure against Some Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 309–318.

[Ala+09] Monjur Alam, Santosh Ghosh, M. J. Mohan, Debdeep Mukhopadhyay, Dipanwita Roy Chowdhury, and Indranil Sengupta. "Effect of glitches against masked AES S-box implementation and countermeasure". In: *IET Inf. Secur.* 3.1 (2009), pp. 34–44.

[And08] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)* Wiley, 2008.

[ANR19] Victor Arribas, Svetla Nikova, and Vincent Rijmen. "Guards in action: First-order SCA secure implementations of KETJE without additional randomness". In: *Microprocess. Microsystems* 71 (2019).

[AP21] Alexandre Adomnicai and Thomas Peyrin. "Fixslicing AES-like Ciphers New bitsliced AES speed records on ARM-Cortex M and RISC-V". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.1 (2021), pp. 402–425.

[AR02] Borivoje Nikolic Anantha P. Chandrakasan and Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Pearson Education, 2002.

[Aro+21] Vipul Arora, Ileana Buhan, Guilherme Perin, and Stjepan Picek. "A Tale of Two Boards: On the Influence of Microarchitecture on Side-Channel Leakage". In: *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*. Ed. by Vincent Grosso and Thomas Pöppelmann. Vol. 13173. Lecture Notes in Computer Science. Springer, 2021, pp. 80–96.

[Ask+22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. "Guarding the First Order: The Rise of AES Maskings". In: *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*. Ed. by Ileana Buhan and Tobias Schneider. Vol. 13820. Lecture Notes in Computer Science. Springer, 2022, pp. 103–122.

[Aum17] Jean-Philippe Aumasson. *Serious cryptography: a practical introduction to modern encryption*. No Starch Press, 2017.

[AZN21] Victor Arribas, Zhenda Zhang, and Svetla Nikova. "LLTI: Low-Latency Threshold Implementations". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 5108–5123. DOI: 10.1109/TIFS.2021.3123527. URL: https://doi.org/10.1109/TIFS.2021.3123527.

[Bak10] R. Jacob Baker. *CMOS - Circuit Design, Layout, and Simulation*. IEEE Press, 2010.

[Bal+12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. "Theory and Practice of a Leakage Resilient Masking Scheme". In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 758–775.

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*. Vol. 8968. Lecture Notes in Computer Science. Springer, 2014, pp. 64–81.

[Bal+15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. "DPA, Bitslicing and Masking at 1 GHz". In: *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 599–619.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by Elisabeth Oswald and Marc

Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 457–485.

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 116–129.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 354–384.

[Bar+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318.

[Bar+20] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations". In: *J. Cryptogr. Eng.* 10.1 (2020), pp. 17–26.

[Bar+21a] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In:

IACR Trans. Cryptogr. Hardw. Embed. Syst. 2021.2 (2021), pp. 189–228.

[Bar+21b] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 189–228.

[Bat+19] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 515–532.

[BC22] Olivier Bronchain and Gaëtan Cassiers. "Bitslicing Arithmetic/-Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 553–588. DOI: 10.46586/TCHES.V2022.I4.553-588. URL: https://doi.org/10.46586/tches.v2022.i4.553-588.

[BCG13] Shivam Bhasin, Claude Carlet, and Sylvain Guilley. "Theory of masking with codewords in hardware: low-weight dth-order correlation-immune Boolean functions". In: *IACR Cryptol. ePrint Arch.* (2013), p. 303.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. "Improved High-Order Conversion From Boolean to Arithmetic Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 22–45.

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. "Analysis and Comparison of Table-based Arithmetic to Boolean Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 275–297.

[Bea+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. "The SIMON and SPECK Families of Lightweight Block Ciphers". In: *IACR Cryptol. ePrint Arch.* (2013), p. 404.

[Bec+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. "Provable Secure Software Masking in the Real-World". In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Ed. by Josep Balasch and

Colin O'Flynn. Vol. 13211. Lecture Notes in Computer Science. Springer, 2022, pp. 215–235.

[Bel+20a] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. "Random Probing Security: Verification, Composition, Expansion and New Constructions". In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. Lecture Notes in Computer Science. Springer, 2020, pp. 339–368.

[Bel+20b] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In: *EUROCRYPT (3)*. Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–341.

[Bel+22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. "IronMask: Versatile Verification of Masking Security". In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 142–160.

[Ben+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. "Deep learning for side-channel analysis and introduction to ASCAD database". In: *J. Cryptogr. Eng.* 10.2 (2020), pp. 163–188.

[Ber08] Daniel J Bernstein. "ChaCha, a variant of Salsa20". In: *Workshop record of SASC*. Vol. 8. 2008, pp. 3–5.

[Bey+21] Tim Beyne, Siemen Dhooghe, Adrián Ranea, and Danilo Sijacic. "A Low-Randomness Second-Order Masked AES". In: *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*. Ed. by Riham AlTawy and Andreas Hülsing. Vol. 13203. Lecture Notes in Computer Science. Springer, 2021, pp. 87–110.

[BH08] Arnaud Boscher and Helena Handschuh. "Masking Does Not Protect Against Differential Fault Attacks". In: *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*. Ed. by Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert. IEEE Computer Society, 2008, pp. 35–40.

[Bil+13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. "Efficient and First-Order DPA Resistant Implementations of Keccak". In: *CARDIS*. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013, pp. 187–199.

[Bil+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "A More Efficient AES Threshold Implementation". In: *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*. Ed. by David Pointcheval and Damien Vergnaud. Vol. 8469. Lecture Notes in Computer Science. Springer, 2014, pp. 267–284.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. "Power Contracts: Provably Complete Power Leakage Models for Processors". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 381–395.

[BO16] Randal E. Bryant and David R. O'Hallaron. *Computer Systems - A Programmer's Perspective, 3rd Edition*. Pearson Education, 2016.

[BS90] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*. Ed. by Alfred Menezes and Scott A. Vanstone. Vol. 537. Lecture Notes in Computer Science. Springer, 1990, pp. 2–21.

[BS91] Eli Biham and Adi Shamir. "Differential Cryptanalysis of Feal and N-Hash". In: *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*. Ed. by Donald W. Davies. Vol. 547. Lecture Notes in Computer Science. Springer, 1991, pp. 1–16.

[BSG23] Fabian Buschkowski, Pascal Sasdrich, and Tim Güneysu. "EasiMask-Towards Efficient, Automated, and Secure Implementation of Masking in Hardware". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*. IEEE, 2023, pp. 1–6.

[Buh+22] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. "SoK: Design Tools for Side-Channel-Aware Implementations". In: *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*. Ed. by Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako. ACM, 2022, pp. 756–770.

[Caf+21] Andrea Caforio, Daniel Collins, Ognjen Glamocanin, and Subhadeep Banik. "Improving First-Order Threshold Implementations of SKINNY". In: *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*. Ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel. Vol. 13143. Lecture Notes in Computer Science. Springer, 2021, pp. 246–267.

[Cas+21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. "Hardware Private Circuits: From Trivial Composition to Full Verification". In: *IEEE Trans. Computers* 70.10 (2021), pp. 1677–1690.

[Cas+24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. "Compress: Reducing Area and Latency of Masked Pipelined Circuits". In: *IACR Cryptol. ePrint Arch.* (2024), p. 1600.

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. "Differential Power Analysis in the Presence of Hardware Countermeasures". In: *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1965. Lecture Notes in Computer Science. Springer, 2000, pp. 252–263.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 188–205.

[Cha+22] Tzu-Hsien Chang, Yen-Ting Kuo, Jiun-Peng Chen, and Bo-Yin Yang. "Secure Boolean Masking of Gimli - Optimization and Evaluation on the Cortex-M4". In: *Information and Communications Security - 24th International Conference, ICICS 2022, Canterbury, UK, September 5-8, 2022, Proceedings*. Ed. by Cristina Alcaraz, Liqun Chen, Shujun Li, and Pierangela Samarati. Vol. 13407. Lecture Notes in Computer Science. Springer, 2022, pp. 376–393.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Che+21] Wei Cheng, Sylvain Guilley, Claude Carlet, Sihem Mesnager, and Jean-Luc Danger. "Optimizing Inner Product Masking Scheme by a Coding Theory Approach". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 220–235.

[Cnu+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. "Does Coupling Affect the Security of Masked Implementations?" In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 1–18. DOI: 10.1007/978-3-319-64647-3_1. URL: https://doi.org/10.1007/978-3-319-64647-3%5C_1.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*. Vol. 7275. Lecture Notes in Computer Science. Springer, 2012, pp. 69–81.

[Cor+13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Springer, 2013, pp. 410–424. DOI: 10.1007/978-3-662-43933-3_21. URL: https://doi.org/10.1007/978-3-662-43933-3%5C_21.

[Cor+15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*. Ed. by Gregor Leander. Vol. 9054. Lecture Notes in Computer Science. Springer, 2015,

pp. 130–149. DOI: 10.1007/978-3-662-48116-5_7. URL: https://doi.org/10.1007/978-3-662-48116-5%5C_7.

[Cor+22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. "High-order Table-based Conversion Algorithms and Masking Lattice-based Encryption". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.2 (2022), pp. 1–40.

[Cor17] Jean-Sébastien Coron. "High-Order Conversion from Boolean to Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 93–114.

[CPW24] Hao Cheng, Daniel Page, and Weijia Wang. "eLIMInate: a Leakage-focused ISE for Masked Implementation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.2 (2024), pp. 329–358.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *CHES*. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference". In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555. DOI: 10.1109/TIFS.2020.2971153. URL: https://doi.org/10.1109/TIFS.2020.2971153.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. "Provably Secure Hardware Masking in the Transition- and Glitch-Robust Probing Model: Better Safe than Sorry". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 136–158.

[Dae17] Joan Daemen. "Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 137–153. DOI: 10.1007/978-3-319-66787-4_7. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_7.

[Das+17] Debayan Das, Shovan Maity, Saad Bin Nasir, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. "High efficiency power side-channel attack immunity using noise injection in attenuated signature domain". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*. IEEE Computer Society, 2017, pp. 62–67.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, 2014, pp. 423–440.

[DF12] Stefan Dziembowski and Sebastian Faust. "Leakage-Resilient Circuits without Computational Assumptions". In: *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*. Ed. by Ronald Cramer. Vol. 7194. Lecture Notes in Computer Science. Springer, 2012, pp. 230–247.

[DFS15a] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. "Making Masking Security Proofs Concrete - Or How to Evaluate the Security of Any Leaking Device". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 401–429.

[DFS15b] Stefan Dziembowski, Sebastian Faust, and Maciej Skorski. "Noisy Leakage Revisited". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 159–188.

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *IEEE Trans. Inf. Theory* 22.6 (1976), pp. 644–654.

[Dho21] Siemen Dhooghe. "Analyzing Masked Ciphers Against Transition and Coupling Effects". In: *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*. Ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel. Vol. 13143. Lecture Notes in Computer Science. Springer, 2021, pp. 201–223.

[Din+14] A. Adam Ding, Liwei Zhang, Yunsi Fei, and Pei Luo. "A Statistical Model for Higher Order DPA on Masked Devices". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 147–169. DOI: 10.1007/

978-3-662-44709-3_9. URL: https://doi.org/10.1007/978-3-662-44709-3%5C_9.

[Dob+18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. "Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures". In: *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*. Ed. by Thomas Peyrin and Steven D. Galbraith. Vol. 11273. Lecture Notes in Computer Science. Springer, 2018, pp. 315–342.

[DOT24] Siemen Dhooghe, Artemii Ovchinnikov, and Dilara Toprakhisar. "StaTI: Protecting against Fault Attacks Using Stable Threshold Implementations". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 229–263.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. "Leakage-Resilient Cryptography". In: *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*. IEEE Computer Society, 2008, pp. 293–302.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DSM22] Siemen Dhooghe, Aein Rezaei Shahmirzadi, and Amir Moradi. "Second-Order Low-Randomness d + 1 Hardware Sharing of the AES". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 815–828.

[Dub+22] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. "ModuloNET: Neural Networks Meet Modular Arithmetic for Efficient Hardware Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 506–556.

[Eva11] Dave Evans. "The internet of Things - How the Next Evolution of the Internet is Changing Everything". In: *Whitepaper, Cisco Internet Business Solutions Group (IBSG)* 1 (2011), pp. 1–12.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 89–120.

[Fel+22] Jakob Feldtkeller, David Knichel, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. "Randomness Optimization for Gadget Compositions in Higher-Order Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 188–227.

[FG05] Wieland Fischer and Berndt M. Gammel. "Masking at Gate Level in the Presence of Glitches". In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 187–200. DOI: 10.1007/11545262_14. URL: https://doi.org/10.1007/11545262%5C_14.

[FH08] Julie Ferrigno and Martin Hlavác. "When AES blinks: introducing optical side channel". In: *IET Inf. Secur.* 2.3 (2008), pp. 94–98. DOI: 10.1049/iet-ifs:20080038. URL: https://doi.org/10.1049/iet-ifs:20080038.

[Fri+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 414–460.

[FSG23] Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. "Challenges and Opportunities of Security-Aware EDA". In: *ACM Trans. Embed. Comput. Syst.* 22.3 (2023), 43:1–43:34.

[Gao+20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. "Share-slicing: Friend or Foe?" In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 152–174.

[Gao+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Hung Pham, and Francesco Regazzoni. "An Instruction Set Extension to Support Software-Based Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 283–325.

[GD23] John Gaspoz and Siemen Dhooghe. "Threshold Implementations in Software: Micro-architectural Leakages in Algorithms". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.2 (2023), pp. 155–179.

[Gen+19] Daniel Genkin, Mihir Pattani, Roei Schuster, and Eran Tromer. "Synesthesia: Detecting Screen Content via Remote Acoustic Side Channels". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 853–869.

[Gho+07] Santosh Ghosh, Monjur Alam, Kundan Kumar, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. "Preventing the side-channel leakage of masked AES S-box". In: *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*. IEEE. 2007, pp. 15–20.

[GHO15] Richard Gilmore, Neil Hanley, and Máire O'Neill. "Neural network based attack on a masked implementation of AES". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE Computer Society, 2015, pp. 106–111.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 1–21.

[Gie+10] Benedikt Gierlichs, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. "Revisiting Higher-Order DPA Attacks:" in: *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*. Ed. by Josef Pieprzyk. Vol. 5985. Lecture Notes in Computer Science. Springer, 2010, pp. 221–234.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[Gig+24a] Barbara Gigerl, Franz Klug, Stefan Mangard, Florian Mendel, and Robert Primas. "Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 309–335.

[Gig+24b] Barbara Gigerl, Florian Mendel, Martin Schläffer, and Robert Primas. "Efficient Second-Order Masked Software Implementations of Ascon in Theory and Practice". In: *In Submission* (2024).

[GIS14] Hendra Guntur, Jun Ishii, and Akashi Satoh. "Side-channel AttacK User Reference Architecture board SAKURA-G". In: *IEEE 3rd Global Conference on Consumer Electronics, GCCE 2014, Tokyo, Japan, 7-10 October 2014*. IEEE, 2014, pp. 271–274.

[GLE15] Zachary N. Goddard, Nicholas LaJeunesse, and Thomas Eisenbarth. "Power analysis of the t-private logic style for FPGAs". In: *IEEE International Symposium on Hardware Oriented Security and Trust,*

HOST 2015, Washington, DC, USA, 5-7 May, 2015. IEEE Computer Society, 2015, pp. 68–71.

[GM11a] Louis Goubin and Ange Martinelli. "Protecting AES with Shamir's Secret Sharing Scheme". In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 79–94.

[GM11b] Tim Güneysu and Amir Moradi. "Generic Side-Channel Countermeasures for Reconfigurable Devices". In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 33–48.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order". In: *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*. Ed. by Helena Handschuh. Vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 251–261.

[Goo+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A testing methodology for side-channel resistance validation". In: *NIST Non-Invasive Attack Testing Workshop*. 2011.

[Gou01] Louis Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 3–15.

[GP99] Louis Goubin and Jacques Patarin. “DES and Differential Power Analysis (The "Duplication" Method)”. In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 158–172.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. “Secure and Efficient Software Masking on Superscalar Pipelined Processors”. In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GPM23a] Barbara Gigerl, Robert Primas, and Stefan Mangard. “Formal Verification of Arithmetic Masking in Hardware and Software”. In: *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part I*. Ed. by Mehdi Tibouchi and Xiaofeng Wang. Vol. 13905. Lecture Notes in Computer Science. Springer, 2023, pp. 3–32.

[GPM23b] Barbara Gigerl, Robert Primas, and Stefan Mangard. “Secure Context Switching of Masked Software Implementations”. In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*. Ed. by Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik. ACM, 2023, pp. 980–992.

[GPM24] Barbara Gigerl, Kevin Pretterhofer, and Stefan Mangard. “Security Aspects of Masking on FPGAs”. In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024*. IEEE, 2024.

[GR16] Dahmun Goudarzi and Matthieu Rivain. “On the Multiplicative Complexity of Boolean Functions and Bitsliced Higher-Order Masking”. In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 457–478.

[GR17] Dahmun Goudarzi and Matthieu Rivain. “How Fast Can Higher-Order Masking Be in Software?” In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. Ed. by Jean-Sébastien

Coron and Jesper Buus Nielsen. Lecture Notes in Computer Science. 2017.

[Gra+19] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet-Moundi. "High-Speed Ring Oscillator based Sensors for Remote Side-Channel Attacks on FPGAs". In: *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*. Ed. by David Andrews, Rene Cumplido, Claudia Feregrino, and Marco Platzner. IEEE, 2019, pp. 1–8.

[Gro] Hannes Gross. *DOM and UMA Protected Hardware Implementations of Ascon*. https://github.com/hgrosz/ascon_dom. Retrieved on April 9th, 2024. URL: https://github.com/hgrosz/ascon_dom.

[Gro+15] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. "Suit up! - Made-to-Measure Hardware Implementations of ASCON". In: *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 2015, pp. 645–652.

[Gro+16a] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. "Bitsliced Masking and ARM: Friends or Foes?" In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*. Vol. 10098. Lecture Notes in Computer Science. Springer, 2016, pp. 91–109.

[Gro+16b] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs". In: *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*. Vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104.

[Gru+21] Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. "DOMREP-An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 4321–4335.

[GSF14] Vincent Grosso, François-Xavier Standaert, and Sebastian Faust. "Masking vs. multiparty computation: how large is the gap for AES?" In: *J. Cryptogr. Eng.* 4.1 (2014), pp. 47–57.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna,*

Austria, August 30 - Sept. 1, 2017. IEEE Computer Society, 2017, pp. 205–212.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 444–461.

[GT02] Jovan Dj. Golic and Christophe Tymen. "Multiplicative Masking and Power Analysis of AES". In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 198–212.

[Gue] Shay Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf, Retrieved on February 23th, 2024.

[Gur+23] Ofek Gur, Tomer Gross, Davide Bellizia, François-Xavier Standaert, and Itamar Levi. "An In-Depth Evaluation of Externally Amplified Coupling (EAC) Attacks - A Concrete Threat for Masked Cryptographic Implementations". In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 70.2 (2023), pp. 783–796.

[Har+03] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. "PINPAS: A tool for power analysis of smartcards". In: *International Conference on Information Security (SEC2003)*. 2003, pp. 453–457.

[Hey+12] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. "Localized Electromagnetic Analysis of Cryptographic Implementations". In: *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*. Ed. by Orr Dunkelman. Vol. 7178. Lecture Notes in Computer Science. Springer, 2012, pp. 231–244.

[HGG20] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. "Applications of machine learning techniques in side-channel attacks: a survey". In: *J. Cryptogr. Eng.* 10.2 (2020), pp. 135–162.

[HH12] David M. Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2012.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. “An AES Smart Card Implementation Resistant to Power Analysis Attacks”. In: *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings*. Ed. by Jianying Zhou, Moti Yung, and Feng Bao. Vol. 3989. Lecture Notes in Computer Science. 2006, pp. 239–252.

[Hos+11] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. “Machine learning in side-channel analysis: a first study”. In: *J. Cryptogr. Eng.* 1.4 (2011), pp. 293–302.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[HS13] Michael Hutter and Jörn-Marc Schmidt. “The Temperature Side Channel and Heating Fault Attacks”. In: *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. Ed. by Aurélien Francillon and Pankaj Rohatgi. Vol. 8419. Lecture Notes in Computer Science. Springer, 2013, pp. 219–235.

[HT19] Michael Hutter and Michael Tunstall. “Constant-time higher-order Boolean-to-arithmetic masking”. In: *J. Cryptogr. Eng.* 9.2 (2019), pp. 173–184.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. “Private Circuits: Securing Hardware against Probing Attacks”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481.

[Jat+20] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, Somitra Kumar Sanadhya, and Donghoon Chang. “Threshold Implementations of ¡tt¿GIFT¡/tt¿: A Trade-Off Analysis”. In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2110–2120.

[Jen+23] Sönke Jendral, Kalle Ngo, Ruize Wang, and Elena Dubrova. “A Single-Trace Message Recovery Attack on a Masked and Shuffled Implementation of CRYSTALS-Kyber”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1587.

[Joh+18] Scoot Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. “Titan: enabling a transparent silicon root of trust for cloud”. In: *Hot Chips: A Symposium on High Performance Chips*. Vol. 194. 2018, p. 10.

[JPS18] Bernhard Jungk, Richard Petri, and Marc Stöttinger. “Efficient Side-Channel Protections of ARX Ciphers”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 627–653.

[JUP24] Darshana Jayasinghe, Brian Udugama, and Sri Parameswaran. “1LUTSensor: Detecting FPGA Voltage Fluctuations using LookUp Tables”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 51–86.

[Ker+22] Maikel Kerkhof, Lichao Wu, Guilherme Perin, and Stjepan Picek. “Focus is Key to Success: A Focal Loss Function for Deep Learning-Based Side-Channel Analysis”. In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Ed. by Josep Balasch and Colin O’Flynn. Vol. 13211. Lecture Notes in Computer Science. Springer, 2022, pp. 29–48.

[Kim+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. “Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.3 (2019), pp. 148–179.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[KK99] Oliver Kömmerling and Markus G. Kuhn. “Design Principles for Tamper-Resistant Smartcard Processors”. In: *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. Ed. by Scott B. Guthery and Peter Honeyman. USENIX Association, 1999.

[KL03] Sung Mo Kang and Yusuf Leblebici. *CMOS digital integrated circuits*. MacGraw-Hill New York, 2003.

[KM22] David Knichel and Amir Moradi. “Low-Latency Hardware Private Circuits”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 1799–1812.

[KM23] David Knichel and Amir Moradi. “Composable Gadgets with Reused Fresh Masks - First-Order Probing-Secure Hardware Circuits with only 6 Fresh Masks”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1141.

[KMC07] Kundan Kumar, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. “Design of a Differential Power Analysis Resistant Masked AES S-Box”. In: *Progress in Cryptology - INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007, Proceedings*. Ed. by K. Srinathan, C. Pandu Rangan, and Moti Yung. Vol. 4859. Lecture Notes in Computer Science. Springer, 2007, pp. 373–383.

[Kni+22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. "Automated Generation of Masked Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 589–629.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113.

[Kön08] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES". In: *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*. Ed. by Tal Malkin. Vol. 4964. Lecture Notes in Computer Science. Springer, 2008, pp. 187–202.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. "Single-Trace Attacks on Keccak". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 243–268.

[Kra+19] Jonas Krautter, Dennis R. E. Gnad, Falk Schellenberg, Amir Moradi, and Mehdi Baradaran Tahoori. "Active Fences against Voltage-based Side Channels in Multi-Tenant FPGAs". In: *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*. Ed. by David Z. Pan. ACM, 2019, pp. 1–8.

[KS20] Pantea Kiaei and Patrick Schaumont. "Domain-Oriented Masked Instruction Set Architecture for RISC-V". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 465.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification". In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Springer, 2020, pp. 787–816.

[LBS19] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. "Reducing a Masked Implementation's Effective Security Order with Setup Manipulations And an Explanation Based on Externally-Amplified Couplings". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 293–317.

[Li+20] Yanbin Li, Ming Tang, Yuguang Li, and Huanguo Zhang. "A pre-silicon logic level security verification flow for higher-order masking schemes against glitches on FPGAs". In: *Integr.* 70 (2020), pp. 60–69.

[Li+22] Yanbin Li, Jiajie Zhu, Yuxin Huang, Zhe Liu, and Ming Tang. "Single-Trace Side-Channel Attacks on the Toom-Cook: The Case Study of Saber". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 285–310.

[Lip+21] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 355–371.

[low19a] lowRISC contributors. *AES HWIP Technical Specification*. https://opentitan.org/book/hw/ip/aes/index.html. Retrieved on 19/4/2023. 2019. URL: https://opentitan.org/book/hw/ip/aes/index.html.

[low19b] lowRISC contributors. *Open Titan*. https://opentitan.org/. Retrieved on March 23th, 2023. 2019. URL: https://opentitan.org/.

[low24] lowRISC contributors. *Ibex: An embedded 32-bit RISC-V CPU core*. https://ibex-core.readthedocs.io/en/latest/index.html. Retrieved on March 4th, 2024. 2024. URL: https://ibex-core.readthedocs.io/en/latest/index.html.

[Lu+21] Xiangjun Lu, Chi Zhang, Pei Cao, Dawu Gu, and Haining Lu. "Pay Attention to Raw Traces: A Deep Learning Architecture for End-to-End Profiling Attacks". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 235–274.

[Man04] Stefan Mangard. "Hardware Countermeasures against DPA ? A Statistical Analysis of Their Effectiveness". In: *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*. Ed. by Tatsuaki Okamoto. Vol. 2964. Lecture Notes in Computer Science. Springer, 2004, pp. 222–235.

[Mat93] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher". In: *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*. Ed. by Tor Helleseth. Vol. 765. Lecture Notes in Computer Science. Springer, 1993, pp. 386–397.

[MBC21] Saurav Maji, Utsav Banerjee, and Anantha P. Chandrakasan. “Leaky Nets: Recovering Embedded Neural Network Models and Inputs Through Simple Power and Timing Side-Channels - Attacks and Defenses”. In: *IEEE Internet Things J.* 8.15 (2021), pp. 12079–12092.

[MDB21] Macarena C. Martinez-Rodriguez, Ignacio M. Delgado-Lozano, and Billy Bob Brumley. “SoK: Remote Power Analysis”. In: *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*. Ed. by Delphine Reinhardt and Tilo Müller. ACM, 2021, 7:1–7:12.

[MDP20] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. “A Comprehensive Study of Deep Learning for Side-Channel Analysis”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 348–375.

[MGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. “Protecting RISC-V against Side-Channel Attacks”. In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 45.

[MM12] Amir Moradi and Oliver Mischke. “Glitch-free implementation of masking in modern FPGAs”. In: *2012 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2012, San Francisco, CA, USA, June 3-4, 2012*. IEEE Computer Society, 2012, pp. 89–95.

[MM17] Thorben Moos and Amir Moradi. “On the Easiness of Turning Higher-Order Leakages into First-Order”. In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 153–170.

[MM21] Thorben Moos and Amir Moradi. “Countermeasures against Static Power Attacks - Comparing Exhaustive Logic Balancing and Other Protection Schemes in 28 nm CMOS -”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 780–805.

[MM22] Nicolai Müller and Amir Moradi. “PROLEAD A Probing-Based Hardware Leakage Detection Tool”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 311–348.

[MMM21] Nicolai Müller, Thorben Moos, and Amir Moradi. “Low-Latency Hardware Masking of PRINCE”. In: *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*. Ed. by Shivam Bhasin and Fabrizio De Santis. Vol. 12910. Lecture Notes in Computer Science. Springer, 2021, pp. 148–167.

[MMR17] Thorben Moos, Amir Moradi, and Bastian Richter. "Static power side-channel analysis of a threshold implementation prototype chip". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1324–1329.

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1297.

[MMT23] Saleh Khalaj Monfared, Tahoura Mosavirik, and Shahin Tajik. "LeakyOhm: Secret Bits Extraction using Impedance Analysis". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 2023, pp. 1675–1689.

[MN07] Mitsuru Matsui and Junko Nakajima. "On the Power of Bitslice Implementation on Intel Core2 Processor". In: *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 121–134.

[Moo+19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. "Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 256–292.

[Moo19] Thorben Moos. "Static Power SCA of Sub-100 nm CMOS ASICs and the Insecurity of Masking Schemes in Low-Noise Environments". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.3 (2019), pp. 202–232.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9.

[Mor+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES". In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 69–88.

[Mor14] Amir Moradi. “Side-Channel Leakage through Static Power - Should We Care about in Practice?” In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 562–579.

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. “Towards Practical Tools for Side Channel Aware Software Engineering: ’Grey Box’ Modelling for Instruction Leakages”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 199–216.

[MP21] Ben Marshall and Dan Page. “SME: Scalable Masking Extensions”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 1416.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. “Side-Channel Leakage of Masked CMOS Gates”. In: *Topics in Cryptology - CT-RSA 2005, The Cryptographers’ Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. Springer, 2005, pp. 351–365.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. “Successfully Attacking Masked AES Hardware Implementations”. In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 157–171.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. “Breaking Cryptographic Implementations Using Deep Learning Techniques”. In: *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Vol. 10076. Lecture Notes in Computer Science. Springer, 2016, pp. 3–26.

[MPW22] Ben Marshall, Dan Page, and James Webb. “MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 175–220. DOI: 10.46586/tches.v2022.i1.175-220. URL: https://doi.org/10.46586/tches.v2022.i1.175-220.

[MQ] Axel Mathieu-Mahias and Michaël Quisquater. “Mixing Additive and Multiplicative Masking for Probing Secure Polynomial Evaluation Methods”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018 ().

[MRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. “Multiplicative Masking for AES in Hardware”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 431–468.

[MSS09] Amir Moradi, Mohammad Taghi Manzuri Shalmani, and Mahmoud Salmasizadeh. “Dual-rail transition logic: A logic style for counteracting power analysis attacks”. In: *Comput. Electr. Eng.* 35.2 (2009), pp. 359–369.

[MSS24] Thorben Moos, Sayandeep Saha, and François-Xavier Standaert. “Prime Masking vs. Faults - Exponential Security Amplification against Selected Classes of Attacks”. In: *IACR Cryptol. ePrint Arch.* (2024), p. 147.

[Mül+22] Nicolai Müller, David Knichel, Pascal Sasdrich, and Amir Moradi. “Transitional Leakage in Theory and Practice Unveiling Security Flaws in Masked Circuits”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.2 (2022), pp. 266–288.

[Mül+23] Nicolai Müller, Sergej Meschkov, Dennis R. E. Gnad, Mehdi B. Tahoori, and Amir Moradi. “Automated Masking of FPGA-Mapped Designs”. In: *33rd International Conference on Field-Programmable Logic and Applications, FPL 2023, Gothenburg, Sweden, September 4-8, 2023*. Ed. by Nele Mentens, Leonel Sousa, Pedro Trancoso, Nikela Papadopoulou, and Ioannis Sourdis. IEEE, 2023, pp. 79–85.

[MWM21] Thorben Moos, Felix Wegener, and Amir Moradi. “DL-LA: Deep Learning Leakage Assessment A modern roadmap for SCA evaluations”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.3 (2021), pp. 552–598.

[Nag+22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. “Riding the Waves Towards Generic Single-Cycle Masking in Hardware”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 693–717.

[Nat01] National Institute of Standards and Technology (NIST). *FIPS-197: Advanced Encryption Standard*. 2001. URL: http://www.itl.nist.gov/fipspubs/.

[Nat02] National Institute of Standards and Technology (NIST). *FIPS-180-2: Secure Hash Standard*. 2002. URL: http://www.itl.nist.gov/fipspubs/.

[New24] NewAE. *CW305 Artix FPGA Target*. https://rtfm.newae.com/Targets/CW305%20Artix%20FPGA/, Retrieved on April 9th, 2024. 2024. URL: https://rtfm.newae.com/Targets/CW305%5C%20Artix%5C%20FPGA/.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*. Vol. 4307. Lecture Notes in Computer Science. Springer, 2006, pp. 529–545.

[NSS22] Yusuke Naito, Yu Sasaki, and Takeshi Sugawara. "Secret Can Be Public: Low-Memory AEAD Mode for High-Order Masking". In: *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part III*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13509. Lecture Notes in Computer Science. Springer, 2022, pp. 315–345.

[OD19] Colin O'Flynn and Alex Dewar. "On-Device Power Analysis Across Hardware Security Domains. Stop Hitting Yourself". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), pp. 126–153.

[ODo14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. Ed. by David Pointcheval. Vol. 3860. Lecture Notes in Computer Science. Springer, 2006, pp. 1–20.

[Pag09] Daniel Page. *A Practical Introduction to Computer Architecture*. Springer, 2009.

[Pap+23] Kostas Papagiannopoulos, Ognjen Glamocanin, Melissa Azouaoui, Dorian Ros, Francesco Regazzoni, and Mirjana Stojilovic. "The Side-channel Metrics Cheat Sheet". In: *ACM Comput. Surv.* 55.10 (2023), 216:1–216:38.

[Pap18] Kostas Papagiannopoulos. "Low Randomness Masking and Shuffling: An Evaluation Using Mutual Information". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), pp. 524–546. DOI: 10.13154/TCHES.V2018.I3.524-546. URL: https://doi.org/10.13154/tches.v2018.i3.524-546.

[Pec08] James K. Peckol. *Embedded systems - a contemporary design tool*. Wiley, 2008.

[Pic+19] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. "The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 209–237.

[Pic+23] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. "SoK: Deep Learning-based Physical Side-channel Analysis". In: *ACM Comput. Surv.* 55.11 (2023), 227:1–227:35.

[Poz+15] Santos Merino Del Pozo, François-Xavier Standaert, Dina Kamel, and Amir Moradi. "Side-channel attacks from static power: when should we care?" In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*. Ed. by Wolfgang Nebel and David Atienza. ACM, 2015, pp. 145–150.

[PP19] Peter Pessl and Robert Primas. "More Practical Single-Trace Attacks on the Number Theoretic Transform". In: *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*. Ed. by Peter Schwabe and Nicolas Thériault. Vol. 11774. Lecture Notes in Computer Science. Springer, 2019, pp. 130–149.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 513–533.

[PR13] Emmanuel Prouff and Matthieu Rivain. "Masking against Side-Channel Attacks: A Formal Security Proof". In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, 2013, pp. 142–159.

[Pra+23] Srinidhi Hari Prasad, Florian Mendel, Martin Schläffer, and Rishub Nagpal. "Efficient Low-Latency Masking of Ascon without Fresh Randomness". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1914.

[PRB09] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. "Statistical Analysis of Second Order Differential Power Analysis". In: *IEEE Trans. Computers* 58.6 (2009), pp. 799–811.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. “Mind the Gap: Towards Secure 1st-Order Masking in Software”. In: *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*. Vol. 10348. Lecture Notes in Computer Science. Springer, 2017, pp. 282–297.

[PWP22] Guilherme Perin, Lichao Wu, and Stjepan Picek. “Exploring Feature Selection Scenarios for Deep Learning-based Side-channel Analysis”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 828–861.

[QS01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards”. In: *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. Ed. by Isabelle Attali and Thomas P. Jensen. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001, pp. 200–210.

[Ram+18] Chethan Ramesh, Shivukumar B. Patil, Siva Nishok Dhanuskodi, George Provelengios, Sebastien Pillement, Daniel E. Holcomb, and Russell Tessier. “FPGA Side Channel Attacks without Physical Access”. In: *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 2018, pp. 45–52.

[Ren+11] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. “A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices”. In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 109–128.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. “Consolidating Masking Schemes”. In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783.

[Roy+15] Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Jean-Luc Danger, and Debdeep Mukhopadhyay. “From theory to practice of private circuit: A cautionary note”. In: *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY,*

USA, October 18-21, 2015. IEEE Computer Society, 2015, pp. 296–303. DOI: 10.1109/ICCD.2015.7357117. URL: https://doi.org/10.1109/ICCD.2015.7357117.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 413–427.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. "Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers". In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 171–188.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[Sas+20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. "Low-Latency Hardware Masking with Application to AES". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 300–326.

[SBM18] Pascal Sasdrich, René Bock, and Amir Moradi. "Threshold Implementation in Software - Case Study of PRESENT". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 227–244.

[Sch+12] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. "Simple Photonic Emission Analysis of AES - Photonic Side Channel Analysis for the Rest of Us". In: *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 41–57.

[Sch+13] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. "Simple photonic emission analysis of AES". In: *J. Cryptogr. Eng.* 3.1 (2013), pp. 3–15.

[Sch+18] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. "An inside job: Remote power analysis attacks on FPGAs". In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. Ed. by Jan Madsen and Ayse K. Coskun. IEEE, 2018, pp. 1111–1116.

[Sch+19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. Lecture Notes in Computer Science. Springer, 2019, pp. 534–564.

[Sch96] Bruce Schneier. *Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition*. Wiley, 1996.

[SD17] Niels Samwel and Joan Daemen. "DPA on hardware implementations of Ascon and Keyak". In: *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*. ACM, 2017, pp. 415–424. DOI: 10.1145/3075564.3079067. URL: https://doi.org/10.1145/3075564.3079067.

[Sha49] Claude E. Shannon. "Communication theory of secrecy systems". In: *Bell Syst. Tech. J.* 28.4 (1949), pp. 656–715.

[She+21a] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. "Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 685–699.

[She+21b] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134.

[Sim+22] Mateus Simoes, Lilian Bossuet, Nicolas Bruneau, Vincent Grosso, Patrick Haddad, and Thomas Sarno. "Self-timed Masking: Implementing Masked S-Boxes Without Registers". In: *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*. Ed. by Ileana Buhan and Tobias Schneider. Vol. 13820. Lecture Notes in Computer Science. Springer, 2022, pp. 146–164.

[Sim+23] Mateus Simões, Lilian Bossuet, Nicolas Bruneau, Vincent Grosso, Patrick Haddad, and Thomas Sarno. "Low-Latency Masking with Arbitrary Protection Order Based on Click Elements". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2023, San Jose, CA, USA, May 1-4, 2023*. IEEE, 2023, pp. 36–47.

[SK23] Soner Seçkiner and Selçuk Köse. "Security Implications of Decoupling Capacitors on Leakage Reduction in Hardware Masking". In: *14th IEEE Latin America Symposium on Circuits and System, LASCAS 2023, Quito, Ecuador, February 28 - March 3, 2023*. IEEE, 2023, pp. 1–4.

[Sko09] Sergei P. Skorobogatov. "Using Optical Emission Analysis for Estimating Contribution to Power Analysis". In: *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009*. Ed. by Luca Breveglieri, Israel Koren, David Naccache, Elisabeth Oswald, and Jean-Pierre Seifert. IEEE Computer Society, 2009, pp. 111–119.

[SKS09] François-Xavier Standaert, François Koeune, and Werner Schindler. "to Compare Profiled Side-Channel Attacks?" In: *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*. Ed. by Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud. Vol. 5536. Lecture Notes in Computer Science. 2009, pp. 485–498.

[SM15] Tobias Schneider and Amir Moradi. "Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations". In: *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 495–513.

[SP20] Stefan Steinegger and Robert Primas. "A Fast and Compact RISC-V Accelerator for Ascon and Friends". In: *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers.*

Ed. by Pierre-Yvan Liardet and Nele Mentens. Vol. 12609. Lecture Notes in Computer Science. Springer, 2020, pp. 53–67.

[Spr+18] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices". In: *IEEE Commun. Surv. Tutorials* 20.1 (2018), pp. 465–488.

[SS16] Peter Schwabe and Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4". In: *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Springer, 2016, pp. 180–194.

[SS22] Kleber Stangherlin and Manoj Sachdev. "Design and Implementation of a Secure RISC-V Microprocessor". In: *IEEE Trans. Very Large Scale Integr. Syst.* 30.11 (2022), pp. 1705–1715.

[Sta+10] François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. "The World Is Not Enough: Another Look on Second-Order DPA". In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*. Ed. by Masayuki Abe. Vol. 6477. Lecture Notes in Computer Science. Springer, 2010, pp. 112–129.

[Sta10] François-Xavier Standaert. "Introduction to Side-Channel Attacks". In: *Secure Integrated Circuits and Systems*. Ed. by Ingrid M. R. Verbauwhede. Integrated Circuits and Systems. Springer, 2010, pp. 27–42.

[Sta18] François-Xavier Standaert. "How (Not) to Use Welch's T-Test in Side-Channel Security Evaluations". In: *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*. Ed. by Begül Bilgin and Jean-Bernard Fischer. Vol. 11389. Lecture Notes in Computer Science. Springer, 2018, pp. 65–79.

[STE15] Aria Shahverdi, Mostafa Taha, and Thomas Eisenbarth. "Silent Simon: A threshold implementation under 100 slices". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*. IEEE Computer Society, 2015, pp. 1–6.

[Sug19] Takeshi Sugawara. "3-Share Threshold Implementation of AES S-box without Fresh Randomness". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 123–145. DOI: 10.13154/tches.v2019.i1.123-145. URL: https://doi.org/10.13154/tches.v2019.i1.123-145.

[SWT01] Dawn Xiaodong Song, David A. Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH". In: *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. Ed. by Dan S. Wallach. USENIX, 2001.

[Tim19] Benjamin Timon. "Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 107–131.

[TKS11] Stefan Tillich, Mario Kirschbaum, and Alexander Szekely. "Implementation and Evaluation of an SCA-Resistant Embedded Processor". In: *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Springer, 2011, pp. 151–165.

[Tri03] Elena Trichina. "Combinational Logic Design for AES SubByte Transformation on Masked Data". In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 236.

[TV04] Kris Tiri and Ingrid Verbauwhede. "A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation". In: *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*. IEEE Computer Society, 2004, pp. 246–251.

[TV05] Kris Tiri and Ingrid Verbauwhede. "Simulation models for side-channel information leaks". In: *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*. Ed. by William H. Joyner Jr., Grant Martin, and Andrew B. Kahng. ACM, 2005, pp. 228–233.

[Udu+22] Brian Udugama, Darshana Jayasinghe, Hassaan Saadat, Aleksandar Ignjatovic, and Sri Parameswaran. "VITI: A Tiny Self-Calibrating Sensor for Power-Variation Measurement in FPGAs". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 657–678.

[VGS14] Nicolas Veyrat-Charvillon, Benoit Gérard, and François-Xavier Standaert. "Soft Analytical Side-Channel Attacks". In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part*

I. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 282–296.

[Voi13] Sorin Voinigescu. *High-Frequency Integrated Circuits*. Cambridge University Press, 2013.

[Wan+23] Chenggang Wang, Jimmy Dani, Shane Reilly, Austen Brownfield, Boyang Wang, and John Marty Emmert. "TripletPower: Deep-Learning Side-Channel Attacks over Few Traces". In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2023, San Jose, CA, USA, May 1-4, 2023*. IEEE, 2023, pp. 167–178.

[Wei+18] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. "I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators". In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 393–406.

[WH11] Neil Weste and David Harris. *CMOS VLSI DEsign - A Circuits and Systems Perspective*. Addison-Wesley, 2011.

[WM18] Felix Wegener and Amir Moradi. "A First-Order SCA Resistant AES Without Fresh Randomness". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 245–262.

[WPP20] Lichao Wu, Guilherme Perin, and Stjepan Picek. "I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis". In: *IACR Cryptol. ePrint Arch.* (2020), p. 1293.

[Wu+23a] Lichao Wu, Amir Ali-pour, Azade Rezaeezade, Guilherme Perin, and Stjepan Picek. "Breaking Free: Leakage Model-free Deep Learning-based Side-channel Analysis". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1110.

[Wu+23b] Lixuan Wu, Yanhong Fan, Bart Preneel, Weijia Wang, and Meiqin Wang. "An automated generation tool of hardware masked S-box: AGEMA^{+}". In: *IACR Cryptol. ePrint Arch.* (2023), p. 831.

[XM88] Guo-Zhen Xiao and James L. Massey. "A spectral characterization of correlation-immune combining functions". In: *IEEE Trans. Inf. Theory* 34.3 (1988), pp. 569–571.

[Yea11] Kim Ho Yeap. *Fundamentals of digital integrated circuit design*. Central Milton Keynes, 2011.

[Yos+20] Kota Yoshida, Takaya Kubota, Shunsuke Okura, Mitsuru Shiozaki, and Takeshi Fujino. "Model Reverse-Engineering Attack using Correlation Power Analysis against Systolic Array Based Neural Network Accelerator". In: *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020*. IEEE, 2020, pp. 1–5.

[You+23] Shih-Chun You, Markus G. Kuhn, Sumanta Sarkar, and Feng Hao. "Low Trace-Count Template Attacks on 32-bit Implementations of ASCON AEAD". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.4 (2023), pp. 344–366.

[Zai+20] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. "Methodology for Efficient CNN Architectures in Profiling Attacks". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 1–36.

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. "PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 391–421.

List of Acronyms

A2B Arithmetic-To-Boolean

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

ARX Addition-Rotation-XOR

ASIC Application-Specific Integrated Circuit

AXI Advanced Extensible Interface

B2A Boolean-To-Arithmetic

CMOS Complementary Metal-Oxide Semiconductor

CPA Correlation Power Analysis

CPU Central Processing Unit

CSR Control and Status Register

DOM Domain-Oriented Masking

DPA Differential Power Analysis

FPGA Field-Programmable Gate Array

HD Hamming distance

HDL Hardware Description Language

HW Hamming weight

IC Integrated Circuit

ILA Independent Leakage Assumption

IoT Internet of Things

ISA Instruction-Set Architecture

ISE Instruction-Set Extension

LSU Load-Store Unit

LUT Lookup table

MOSFET Metal-Oxide Semiconductor Field Effect Transistor

NIST National Institute of Standardization and Technology

NMOS N-channel Metal-Oxide Semiconductor

NTT Number Theoretic Transform

OS Operating System

PCB Printed Circuit Board

PMOS P-channel Metal-Oxide Semiconductor

PQC Post-Quantum Cryptography

RAM Random-Access Memory

RNG Random Number Generator

RTL Register Transfer Level

RTOS Real-Time Operating Systems

ROM Read-Only Memory

SCA Side-Channel Analysis

SoC System on a Chip

SPA Simple Power Analysis

TI Threshold Implementation

TVLA Test Vector Leakage Assessment

List of Contributions

Main Publications

The following publications are scientific contributions of the author that are discussed in this thesis.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[Gig+24a] Barbara Gigerl, Franz Klug, Stefan Mangard, Florian Mendel, and Robert Primas. "Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 309–335.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors". In: *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13091. Lecture Notes in Computer Science. Springer, 2021, pp. 3–32.

[GPM23a] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Formal Verification of Arithmetic Masking in Hardware and Software". In: *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part I*. Ed. by Mehdi Tibouchi and Xiaofeng Wang. Vol. 13905. Lecture Notes in Computer Science. Springer, 2023, pp. 3–32.

[GPM23b] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure Context Switching of Masked Software Implementations". In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*. Ed. by Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik. ACM, 2023, pp. 980–992.

[GPM24] Barbara Gigerl, Kevin Pretterhofer, and Stefan Mangard. “Security Aspects of Masking on FPGAs”. In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024*. IEEE, 2024.

Further Publications

The following list of publications comprises collaborations and contributions of the author, which are not part of this thesis.

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. “Power Contracts: Provably Complete Power Leakage Models for Processors”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 381–395.

[Cas+24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. “Compress: Reducing Area and Latency of Masked Pipelined Circuits”. In: *IACR Cryptol. ePrint Arch.* (2024), p. 1600.

[Gig+24b] Barbara Gigerl, Florian Mendel, Martin Schläffer, and Robert Primas. “Efficient Second-Order Masked Software Implementations of Ascon in Theory and Practice”. In: *In Submission* (2024).

[Nag+22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. “Riding the Waves Towards Generic Single-Cycle Masking in Hardware”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 693–717.