EASIMASK- Towards Efficient, Automated, and Secure Implementation of Masking in Hardware

Fabian Buschkowski, Pascal Sasdrich, Tim Güneysu
Ruhr-University Bochum
{fabian.buschkowski, pascal.sasdrich, tim.gueneysu}@rub.de

***Abstract*—Side-Channel Analysis (SCA) is a major threat to implementations of mathematically secure cryptographic algorithms. Applying masking countermeasures to hardware-based implementations is both time-consuming and error-prone due to side-effects buried deeply in the hardware design process.**

As a consequence, we propose our novel framework EASIMASK in this work. Our semi-automated framework enables designers that have little experience with hardware implementation or physical security and the application of countermeasures to create a securely masked hardware implementation from an abstract description of a cryptographic algorithm. Its design-flow dismisses the developer from many challenges in the masking process of hardware implementations, while the generated implementations match the efficiency of hand-optimized designs from experienced security engineers. The modular approach can be mapped to arbitrary instantiations using different languages and transformations. We have verified the functionality, security, and efficiency of generated designs for several state of the art symmetric cryptographic algorithms, such as Advanced Encryption Standard (AES), Keccak, and PRESENT.

***Index Terms*—cryptography, hardware, masking**

I. INTRODUCTION

The discovery of Side-Channel Analysis (SCA) by Paul Kocher [1], [2] was a disruptive moment for academia and industry alike. Through close observation of electronic devices and their physical characteristics during execution of cryptographic implementations, adversaries could extract secret internals and information that leak unintentionally [1]–[3].

Consequently, appropriate countermeasures were investigated, under which *masking* [4] is considered most promising due to its sound theoretical foundations. Still, correct and secure implementation of masked cryptographic algorithms is a mostly manual, laborious, and error-prone task. Especially in hardware, the security guarantees of masked designs may be undermined by unintentional *physical effects* [5]–[7].

As a result, a variety of different hardware masking schemes [8]–[11] have been presented in the course of time, to propose optimized solutions with respect to area consumption, latency increase, and demand of fresh randomness.

Notably, most hardware masking schemes focus on the correct design and implementation of atomic components, often considered as *gadgets*. Unfortunately, composition of securely masked gadgets is non-trivial and does not necessarily lead to secure constructions. Therefore, most modern gadgets additionally adhere to secure composability notions such as Probe-Isolating Non-Interference (PINI) [12]. Still, secure masking of unprotected circuits remains a manual and error-prone process. For this purpose, Knichel et al. recently presented AGEMA [13], a novel post-synthesis masking tool, allowing automated construction of securely masked gate-level netlists using selected and pre-synthesized masked gadgets.

Although AGEMA was designed to assist engineers and practitioners in reducing the required experience in hardware security, it still demands deep knowledge on the design processes and flows. This is especially true as the optimization of a circuit during synthesis does not necessarily result in an optimized masked circuit but, in a worst case, reduces the efficiency by introducing additional overhead.

Contributions: In this work we present a first framework to semi-automatically generate SCA-protected hardware implementations of round-based symmetric cryptosystems. Our concept is based on a pre-synthesis step that processes an abstract description of a cryptographic component. While our approach is fully generic, wide applicability is achieved by the integration of pre-optimized protected components through an associated Intellectual Property (IP)-library. Several different designs can be created from a single abstract description through the variation of a number of parameters, including time-area trade-offs, security order, randomness generation and, optionally, the cryptographic mode of operation. We demonstrate the successful generation of protected cryptographic designs with several case-studies, providing the corresponding performance figures and side-channel experiments. Our source code and the IP-library are available on GitHub.

Outline: The remainder of this work is organized as follows: While we give background information on SCA and masking in Section II, the concept of our framework and its implementation with our choices for design representations and transformations is presented in Section III. In Section IV, we verify the functionality, security, and efficiency of generated designs for a wide range of state of the art symmetric cryptographic algorithms. Finally, we conclude our work in Section V.

II. BACKGROUND

A. Side-Channel Analysis

SCA is a powerful attack vector on cryptographic algorithms as it lowers the problem space of guessing a secret using a divide-and-conquer approach. Exploiting the power consumption of a device is the most popular type of SCA, but many of the attacks and countermeasures work similarly for other side channels. While the simplest form of SCA only requires capturing and investigating a single trace, more powerful attacks, e.g., Differential Power Analysis (DPA) or

978-3-9819263-7-8/DATE23/© 2023 EDAA

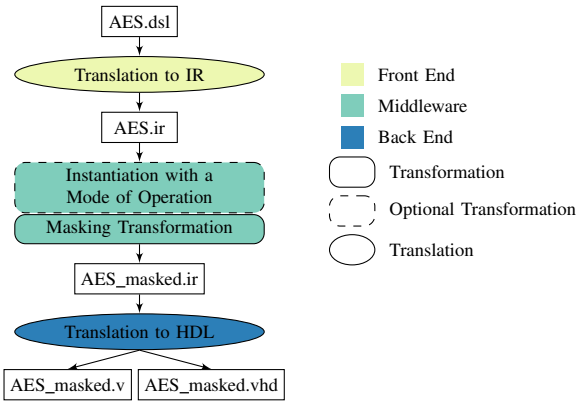


Fig. 1: Design flow of our framework

Correlation Power Analysis (CPA) require capturing up to millions of traces and the following analysis using statistical means such as the correlation coefficient.

B. Boolean Masking

By randomizing intermediate values, masking aims to make the physical characteristics of a device independent of the processed secret data. In Boolean masking, a secret value $x \in \mathbb{F}_n$ is split into $s \geq 2$ shares $(x_1, ..., x_s) \in \mathbb{F}_n$ such that $\bigoplus_{i=1}^{s} x_i = x$, where s depends on the desired security order and the used masking scheme. This is usually achieved by sampling $x_i \xleftarrow{\$} \mathbb{F}_n$ for $1 \leq i \leq s-1$, and setting $x_s = x \oplus \bigoplus_{i=1}^{s-1} x_i$. While linear functions operate on each share individually, non-linear functions use multiple shares while avoiding to unmask the secret value. Non-linear functions often have additional register stages and need fresh random bits every clock cycle, whose amount increases with the security order. Several schemes describing the process of masking non-linear functions, such as Domain-Oriented Masking (DOM) [11], have been proposed in recent years.

III. Framework Description

In this section, we introduce the conceptual principles and the specific implementation of our framework. We opted to follow a classical process structure as depicted in Figure 1. Given an abstract description of a cryptographic algorithm, the front end translates the design into the Intermediate Representation (IR). IR-to-IR transformations, namely masking and, optionally, the instantiation with a mode of operation, modify the design, before it is finally translated to a securely masked design in a Hardware Description Language (HDL) in the back end. This approach has two major advantages: (1) Input and output languages can be changed without having to modify the transformations, and (2) an arbitrary amount of transformations can be added into the design flow.

A. Design Representations

In order to appropriately represent the design at different levels of abstraction and perform necessary translation and transformation processes, our framework requires multiple design representations and description languages, as discussed in the following paragraphs.

Feature	DSL	HCL	HDL
Algorithm Description			
Functions/Modules	✓	✓	✓
(Un-)Signed Types	✓	✓	✓
Logic Functions	✓	✓	✓
Arithmetic Functions	✓	✓	✓
Arrays	(✓)		
Table Lookups and Permutations	(✓)		
Hardware			
Registers		✓	✓
Attributes for synthesis tools		✓	✓

TABLE I: Basic features required for DSLs, HCLs, and HDLs

1) Domain Specific Language: To dismiss a designer from the often tedious and error-prone task of implementing a cryptographic scheme in hardware, our design flow starts with an abstract, high-level description of the algorithm.

Domain Specific Languages (DSLs) are an ideal choice for the abstract description as they are tailored for the use in a specific area. Equipped with dedicated features that support the use in their respective domain, DSLs generally have a higher level of abstraction compared to general-purpose languages such as C or Java, which allows a designer to describe a cryptographic algorithm with few lines of code.

We decided on several minimal criteria, shown in Table I, that a DSL has to fulfill to be able to describe a cryptographic algorithm: The ability to define functions or modules, the support for (un-)signed data types, and logic and arithmetic functions. Arrays of data types and the ability to express table lookups and permutations are optional basic features that can be useful when describing parts of cryptographic algorithms, e.g., S-Boxes. We identified three candidates that fulfill these basic requirements, namely CRYPTOL [14], USUBA [15], and CAO [16], while other DSLs such as SQL or regular expressions did not meet these requirements. Guiding our decision, we decided on additional features, shown in Table II, that improve the usability for the user, such as support for GF arithmetic (e.g., for Advanced Encryption Standard (AES)) and user-defined types, or aid the translation in the front end, like existing translations. From the list of existing candidates, CRYPTOL was identified as most suitable choice since it has a translation to VHDL and Verilog which we can use for the translation from CRYPTOL to our IR, and the possibility to execute code. This allows a user to verify the correctness of the specification against test values and ensure that it contains no errors before continuing in the design flow.

2) Intermediate Representation: The IR shares the basic requirements of the DSL as it also has to be able to describe cryptographic algorithms. Additionally, as shown in Table I, the IR has to be able to describe registers, which will play an important role during masking, and support the addition of annotations, which are added by the front end to transport metadata to the transformations in the middleware or synthesis attributes into the HDL code.

We decided to consider Hardware Construction Languages (HCLs) for our IR, as these languages allow to express hard-

Feature	CRYPTOL	USUBA	CAO
Usability			
Support for GF Arithmetic	✓	✗	✓
Conditional Statements	✓	✗	✓
User-defined types	✓	✗	✓
Bit Manipulation	✓	✗	✓
Code Base			
Active Community	✓	✓	✗
Extensibility	✓	✓	✓
Code Available	✓	✓	✗
Translation	VHDL, Verilog	C	C
Code Execution	✓	✗	✗

TABLE II: Extended features of DSLs

Feature	SPINALHDL	CHISEL	CLASH
Functionality			
Inclusion of IP	VHDL, Verilog	VHDL	✗
FSM	✓	✓	✓
Counter	✓	✓	✓
Code Base			
Active development	✓	✓	✓
Extensibility	✓	✓	✓
Translation	VHDL, Verilog	VHDL, Verilog	VHDL, Verilog

TABLE III: Extended features of HCLs

ware constructs just like VHDL or Verilog, but offer additional functionality features to define Finite State Machines (FSMs) and counters. We investigated a total of five HCLs, of which three, namely SPINALHDL [17], CHISEL [18] and CLASH [19], met the basic requirements, while MYHDL and NMIGEN were no suitable candidates due to their inability to express synthesis attributes. Again, we decided on additional features, shown in Table III, that enhance the functionality of the language. It turned out that both SPINALHDL and CHISEL offer all identified features. We decided to use SPINALHDL since it enables the inclusion of VHDL and Verilog IP, which is useful for integration of premasked (non-linear) building blocks in the masking transformation.

3) Hardware Description Language: Our framework outputs a representation of the cryptographic algorithm in an HDL. Apart from the basic features named in Table I, it is crucial that the output of our framework can be processed by standard synthesis and place&route tools. Furthermore, the HDL has to support attributes that prevent the removal of security critical parts during synthesis.

As the standard HDLs VHDL and Verilog fulfill the aforementioned criteria and a translation from SPINALHDL to these languages already exists, we decided to target VHDL and Verilog in our framework.

B. Processes

For the processes, we specifically distinguish between *translations*, which translate the design from one representation to another, and *transformations* that modify the design on the IR, e.g., to add new features. Figure 2 shows the necessary user inputs for the different translations and transformations used in our framework with default values in bold.

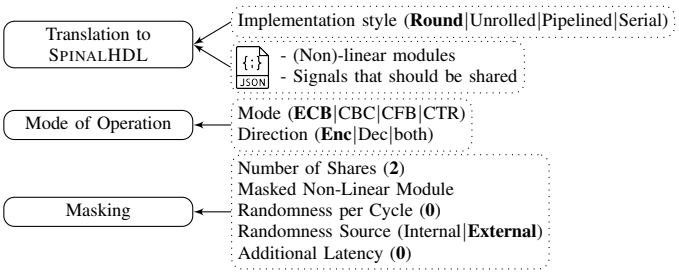


Fig. 2: Required user inputs (right) for the used translations and transformations (left) in our framework

1) Front End: In the front end, the abstract description of the cryptographic algorithm is translated from the DSL to the IR. Each function is translated into a separate module in the IR. A `keep_hierarchy` annotation is added to prevent the synthesis tool from removing modules, which could harm the security of the design. The parameters of a function are translated to input and output ports of its corresponding module. Local variables of a function are translated to internal signals in the IR, for which the front end has to decide whether they should be defined as registers or as simple wires.

This is decided according to a predefined implementation style chosen by the user. The user can choose between a *round-based*, *unrolled*, *pipelined*, and *serial* implementation, with *round-based* being the default option. In this case, the output of the round function and the round counter, and, if necessary, the result of the key schedule, are stored in a register and updated after every round, while all other signals are implemented as simple wires. For an *unrolled* design, all signals are defined as wires, resulting in a latency of just one clock cycle. The *pipelined* implementation style implements the state and round key in separate registers for every round. A *serial* implementation has the same registers as a *round-based* one, but modules, e.g., S-Boxes, are shared and reused across the design. This results in a higher latency, but lower area compared to the other implementation styles. All variants can be processed by the masking transformation.

The user also specifies via JSON which CRYPTOL functions are linear or non-linear, and whether parameters of the top-level function need to be shared. Usually, S-Boxes and adders are the only non-linear operations in a symmetric cryptographic algorithm, and all inputs apart from the control logic are shared. The key of an encryption algorithm can be unshared to reduce the initial randomness. The information is added as an annotation in the SPINALHDL code by the front end.

2) Middleware: In the middleware, the design can be extended and modified by IR-to-IR transformations. Every transformation has to output syntactically and semantically correct code to ensure that eventual following transformations are able to process the code. A transformation may consist of several sub-transformations that must be executed in sequence and in the right order to produce a functionally correct design.

For our framework, the central transformation is *masking*, which transforms the original design into a securely masked design. Three components of the design are addressed in

separate sub-transformations: Linear and non-linear functions, and the controlling logic, e.g., the FSM. We discuss the three sub-transformations in their order of execution.

a) Masking linear operations: The first transformation masks wires, registers, and linear operations by duplicating them according to the number of shares, and then connecting them to each other. The number of shares specified by the user has to match the number of shares in the non-linear module that is to be used. Otherwise, the resulting design will not be functionally correct. Annotations indicate whether input and output signals have to be shared or not, and our framework automatically infers the sharing of internal signals by propagating this information through the design. If multiple signals are combined and at least one of them is shared, the result also needs to be shared. A `keep` annotation is added to all shares and prevents the removal of signals during synthesis.

b) Masking non-linear operations: While generic approaches to mask non-linear operations exist, these approaches use the Algebraic Normal Form (ANF) of a function, which is unavailable at the pre-synthesis stage. Therefore, we decided to replace entire non-linear modules (e.g., S-Boxes or adders) with handcrafted masked solutions that have been used in literature to create secure implementations. We provide an IP-library with securely masked modules that users can choose from. The IP-library includes S-Boxes of different type and security order from various cryptographic algorithms, including AES, PRESENT, Prince, Keccak, and others. Further S-Boxes can be added to the IP-library in the future.

To provide fresh randomness to the non-linear modules, users can choose between an external source of randomness, such as a Pseudorandom Number Generator (PRNG) running on a different device, and a PRNG integrated into the design. In the first case, an input port for the randomness with the appropriate size is created and connected to the non-linear modules. Otherwise, a PRNG from our IP-library, which includes Keccak, AES, and others, can be chosen. The outputs of the PRNG are connected to the non-linear modules, and an additional input port for a seed is added to the main module and connected to the PRNG. The user can additionally specify a number of clock cycles that the PRNG should run for before the cryptographic algorithm is executed to ensure that the generated randomness is as close to uniform as possible. If no interval is specified, the execution of the cryptographic algorithm and the PRNG are started at the same time.

c) Modifying the control logic: In the final masking transformation, the controlling logic of the design is modified to account for the additional latency of masked non-linear modules, which has to be specified by the user. If the specified latency is zero, this transformation is skipped.

The round counter of the design has to be kept synchronized with the rounds with higher latency. To achieve this, a second counter is added and incremented every cycle. Once it reaches the specified additional latency, it is reset and the round counter is incremented. If a signal is combined with an output of a non-linear module with higher latency, our framework automatically inserts additional register stages to keep it synchronized. The number of additional register stages is equal to the additional latency, e.g., if the non-linear module takes one additional clock cycle, one register stage is added.

Users of our framework can write new IR-to-IR transformations and insert them before or after the masking transformation to further modify the design. Transformations are written in Scala and inserted as so-called *phases* in the SpinalHDL back end, where they modify the current design. There is no limit on the number of additional transformations. The outputs of a transformation added before the masking will be processed and masked by the masking step. It is therefore important that the annotations made by the front end are not removed. On the other hand, a transformation added after the masking step must not remove or combine shares of signals, duplicated modules or registers, or modify the non-linear components in any way.

Our framework offers an optional pre-masking transformation to instantiate a mode of operation around a block cipher. While it could be implemented by the user in the Cryptol specification, the transformation frees the user of this task and ensures the correct implementation of the mode of operation. The supported modes are Electronic Code-Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Counter Mode (CTR). The user specifies the desired mode and whether the input is to be encrypted or decrypted, or if a combined architecture is needed. The default setting is the encryption with the ECB mode. The instantiated mode of operation will be masked by the subsequent masking transformation. As a result, the initialization vector for CBC, CFB, and CTR mode are implemented as shared, and internal XOR operations will be instantiated multiple times. In the CTR mode, the shared arithmetic addition necessary for the counter is implemented using a Kogge-Stone adder that uses DOM AND-gadgets for its non-linear operations. The addition is performed in parallel to the processing of the current data block.

3) Back End: In the back end, the design is translated from the IR to the target HDL. Extended features of the IR are translated to equivalent HDL code. Annotations intended for the synthesis tool such as `keep` are translated to HDL attributes, while the other annotations in the IR code, e.g., on the sharing of signals, are ignored.

As the input to the back end is plain SpinalHDL code, we can use the default translations to VHDL and Verilog offered by SpinalHDL, which also translate annotations for the synthesis tool to the target languages.

C. Discussion

Starting with an abstract description in Cryptol enables users unfamiliar with hardware to create an implementation of a cryptographic algorithm in hardware. While it is necessary to get used to Cryptol and create an algorithm description in it, this can be easily achieved due to the limited scope and simple syntax of Cryptol.

The masking transformation can be influenced by a variety of parameters that have a direct impact on the efficiency and security of the generated designs, allowing users to create

multiple designs with different properties from a single specification. This is further supported by our open-source IP-library that offers a wide range of masked and optimized non-linear modules and allows to increase the range of obtainable designs.

With the output of VHDL and Verilog code in the end, we can integrate our framework into the standard hardware design tool flow. Annotations added by our framework ensure that the masking countermeasures are not removed during synthesis.

IV. Case Studies

To show the wide range of applicability of our tool, we applied it to CRYPTOL specifications of multiple cryptographic algorithms, and evaluated the functionality and security of the generated masked hardware implementations. We chose AES as the most popular and commonly used block cipher, and PRESENT as a popular lightweight alternative to AES. In order to also cover the area of hash functions, we additionally evaluated our tool for an implementation of Keccak.

To verify the security of the designs, we performed experimental evaluations on a Field-Programmable Gate Array (FPGA). Our measurements were done on a SAKURA-G [20] evaluation board which is equipped with a Spartan-6 FPGA. The power consumption of the device is measured by a digital sampling oscilloscope at a frequency of 1.25 GS/s, while the FPGA is driven by a 4 MHz clock. The measured traces are quantized with a 16-bit resolution. In all experiments, we followed the well-established Test Vector Leakage Assessment (TVLA) approach [21] and performed fixed-plaintext versus random-plaintext t-tests at first and second order using 50 million traces. If the t-values always remain within the interval of ± 4.5, the device is assumed to have no detectable leakage and considered secure (with high confidence). The measurements, displayed in Figure 3, are discussed in the following.

A. PRESENT

For our evaluation of PRESENT-128, we used a *round-based* implementation and the first-order secure TI S-Box [22] from our IP-library, which has an additional latency of one clock cycle compared to the unmasked S-Box. Figure 3d displays the results of the measurement. The first-order t-test in Figure 3b shows no leakage, indicating that the design was securely masked by our tool, while the second-order t-test in Figure 3c expectedly detects multiple leakage points.

B. AES

For AES-128, we selected a *serial* implementation that only uses one S-Box. To verify that including non-linear modules in form of VHDL files works correctly, we passed the VHDL files of the first-order secure DOM S-Box [23] with five stages and pipelining activated to the masking transformation. A Keccak instance running on the target FPGA produces the necessary fresh randomness. The PRNG runs for 10 clock cycles before starting the AES design to ensure the quality of the generated randomness. Figure 3h shows the results of the measurements. As expected, no first-order leakage is detected. We additionally created a second-order *serial* and a first-order *round-based* implementation and validated their side-channel security.

Algorithm	Area			Latency	Ref.
	[LUT]	[FF]	[kGE]	[cycles]	
Keccak	11 648	9 634	114.1	49	new
Keccak	11 416	9 610	111.8	48	[25]
AES-128 1st order serial	684	612	7.8	201	new
AES-128 1st order serial	646	584	7.6	200	[11]
AES-128 2nd order serial	1 157	1 024	13.1	201	new
AES-128 2nd order serial	1 080	946	12.8	200	[11]
AES-128 1st order round-based	5 124	3 824	62.6	51	new
AES-128 1st order round-based			$\approx$ 60.8	50	[11]
PRINCE	1 445	1 276	11.9	25	new
PRINCE			11.5	24	[26]
PRESENT-128	1 390	1 173	11.3	63	new
PRESENT-128			$\approx$ 10.5	62	[22]

TABLE IV: Synthesis results

C. Keccak

In our final case study, we evaluated a first-order secure implementation of the Keccak hash function. We used a *round-based* implementation with the first-order secure DOM Keccak S-Box from GitHub [24]. Figure 3l displays the results of our measurement, showing no first-order leakage.

D. Implementation Results

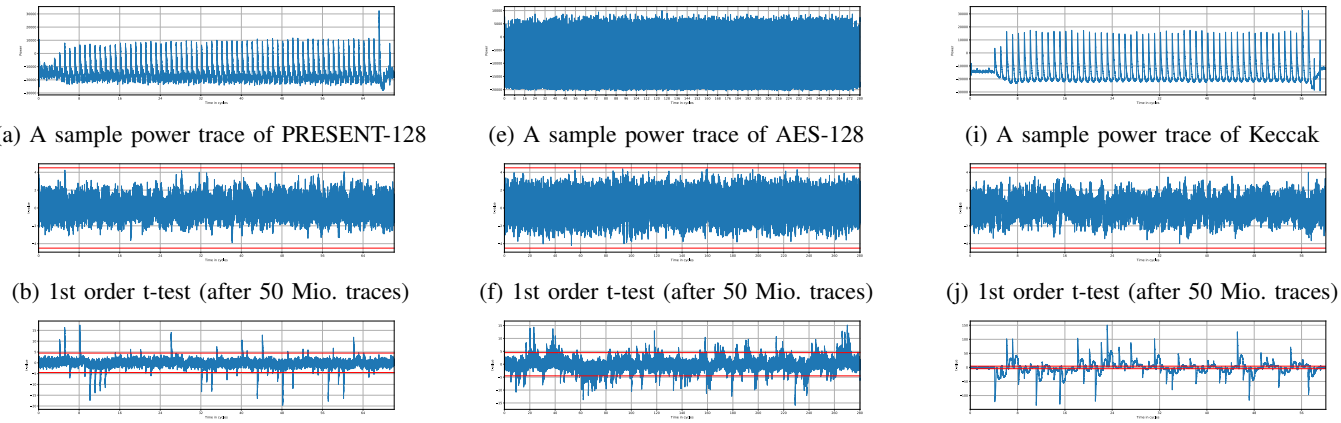
We synthesized all implementations to a Xilinx Spartan-6 FPGA using Xilinx ISE 14.7, or for an Application-Specific Integrated Circuit (ASIC) using Synopsis Design Compiler and the UMC 180 standard cell library. The synthesis results can be seen in Table IV.

To compare our Keccak implementation with the one by Gross et al. [25], we used their publicly available VHDL sources [24] and adapted the parameters in the code to match our implementation. The synthesis reports that our implementation is less than 2% larger than the handcrafted design. For AES, we synthesized the DOM implementations available on GitHub [23]. The synthesis results indicate that our implementations are marginally larger by less than 3%. For the round-based designs, we estimated the area requirements of the reference implementation, which is around 3% smaller than the design by EASIMASK. Finally, we compare our implementation of PRESENT with the one by Poschmann et al. [22]. Using the same approximation as for the AES implementation, our implementation is around 8% larger.

The synthesis results show that the designs created by EASIMASK come close to handmade designs for the area and latency, barring an overhead of 3% or less in terms of area. This overhead is mainly introduced in the front end, since the hardware architecture is created in a generic, unoptimized way.

V. Conclusion

In this work, we introduced our open-source framework EASIMASK, enabling users to create a securely masked hardware implementation of a cryptographic algorithm from an abstract high-level specification. Our framework requires little experience in the fields of hardware design and physical security from a user, making it an attractive tool to use for



(a) A sample power trace of PRESENT-128 (e) A sample power trace of AES-128 (i) A sample power trace of Keccak

(b) 1st order t-test (after 50 Mio. traces) (f) 1st order t-test (after 50 Mio. traces) (j) 1st order t-test (after 50 Mio. traces)

(c) 2nd order t-test (after 50 Mio. traces) (g) 2nd order t-test (after 50 Mio. traces) (k) 2nd order t-test (after 50 Mio. traces)

(d) Measurements for PRESENT-128 (h) Measurements for AES-128 (l) Measurements for Keccak

Fig. 3: Measurement results

practitioners in these fields. The modular structure of our framework allows users to adapt it to their needs by exchanging languages and transformations of the design flow.

The masking in our framework can be influenced by a number of parameters, allowing to create several different masked designs from a single algorithm specification. The integration of optimized masked S-Boxes enables our tool to generate implementations that are on par with handcrafted hardware implementations in terms of size and speed. With the optional instantiation of a mode of operation around a block cipher, our framework allows to create an implementation that is suitable for use in a real-world setting where large amounts of data have to be encrypted and decrypted in a secure way.

In our case studies we successfully validated that our tool generates correct circuits with automatically generated masking countermeasures that are capable to withstand practical power side-channel attacks.

VI. Acknowledgments

The work in this paper has been supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA - 390781972) and Emmy-Noether-Project CAVE (510964147), and through the project VE-HEP (16KIS1345) supported by the German Federal Ministry of Education and Research (BMBF).

References

[1] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO 1996*, 1996.

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO 1999*, 1999.

[3] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM Side-Channel(s)," in *CHES 2002*, 2002.

[4] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," in *CRYPTO 1999*, 1999.

[5] T. Moos, A. Moradi, T. Schneider, and F. Standaert, "Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed," *IACR TCHES*, 2019.

[6] S. Mangard, T. Popp, and B. M. Gammel, "Side-Channel Leakage of Masked CMOS Gates," in *CT-RSA 2005*, 2005.

[7] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, "On the Cost of Lazy Engineering for Masked Software Implementations," in *CARDIS 2014*, 2014.

[8] Y. Ishai, A. Sahai, and D. A. Wagner, "Private Circuits: Securing Hardware against Probing Attacks," in *CRYPTO 2003*, 2003.

[9] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold Implementations Against Side-Channel Attacks and Glitches," in *ICICS 2006*, 2006.

[10] H. Groß, S. Mangard, and T. Korak, "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order," in *CT-RSA 2017*, 2017.

[11] ——, "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order," in *TIS@CCS 2016*. ACM, 2016.

[12] G. Cassiers and F. Standaert, "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference," *IEEE TIFS*, 2020.

[13] D. Knichel, A. Moradi, N. Müller, and P. Sasdrich, "Automated Generation of Masked Hardware," *IACR TCHES*, 2022.

[14] J. R. Lewis and B. Martin, "CRYPTOL: High assurance, retargetable crypto development and validation," Oct. 2003.

[15] D. Mercadier, P. Dagand, L. Lacassagne, and G. Muller, "Usuba: Optimizing & Trustworthy Bitslicing Compiler," in *WPMVP@PPoPP 2018*. ACM, 2018.

[16] M. Barbosa, R. Noad, D. Page, and N. Smart, "First steps toward a cryptography-aware language and compiler." *IACR Cryptology ePrint Archive*, vol. 2005, p. 160, 01 2005.

[17] "SpinalHDL," 2022. [Online]. Available: https://github.com/SpinalHDL/SpinalHDL

[18] "Constructing Hardware in a Scala Embedded Language (Chisel)," 2022. [Online]. Available: https://www.chisel-lang.org/

[19] "Clash - A modern, functional, hardware description language," 2022. [Online]. Available: https://clash-lang.org/

[20] S. Lab, "SAKURA hardware security project," 2014. [Online]. Available: https://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html

[21] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side channel resistance," 2011.

[22] A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling, "Side-channel resistant crypto for less than 2, 300 GE," *J. Cryptol.*, vol. 24, no. 2, pp. 322–345, 2011. [Online]. Available: https://doi.org/10.1007/s00145-010-9086-6

[23] H. Gross, "aes-dom," 2016. [Online]. Available: https://github.com/hgrosz/aes-dom

[24] D. Schaffenrath, "keccak_dom," 2017. [Online]. Available: https://github.com/hgrosz/keccak_dom

[25] H. Gross, D. Schaffenrath, and S. Mangard, "Higher-Order Side-Channel Protected Implementations of KECCAK," in *DSD 2017*, 2017.

[26] A. Rezaei Shahmirzadi and A. Moradi, "Re-Consolidating First-Order Masking Schemes: Nullifying Fresh Randomness," *IACR TCHES*, 2020.