IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 155–179. DOI:10.46586/tches.v2023.i2.155-179

Threshold Implementations in Software: Micro-architectural Leakages in Algorithms

John Gaspoz and Siemen Dhooghe

imec-COSIC, ESAT, KU Leuven, Belgium
name.lastname@esat.kuleuven.be

Abstract. This paper provides necessary properties to algorithmically secure first-order maskings in scalar micro-architectures. The security notions of threshold implementations are adapted following micro-processor leakage effects which are known to the literature. The resulting notions, which are based on the placement of shares, are applied to a two-share randomness-free PRESENT cipher and `Keccak-`f. The assembly implementations are put on a RISC-V and an ARM Cortex-M4 core. All designs are validated in the glitch and transition extended probing model and their implementations via practical lab analysis.

Keywords: Masking, Micro-architectures, Side-channel Analysis, Probing Security

1 Introduction

In their seminal work, Kocher *et al.* [KJJ99] demonstrated that cryptographic primitives —although mathematically secure in a black-box setting— can suffer from attacks once deployed in the real world such as in embedded devices. Side channel analysis enables an adversary to recover secret data by observing physical characteristics (*e.g.* power consumption) from such a device. In an attempt to prevent these attacks, various countermeasures were designed. Masking is one of the most prevalent countermeasures which aims to make computations independent from the input and prevent its direct power consumption analysis. To do so, a d^{th}-order masking splits sensitive data into $d+1$ random shares such that observation of up to d shares does not provide enough information to recover the original value. Hence, provided certain assumptions such as independent leakage, masking countermeasures are theoretically secure.

However, the assumptions on which masking relies to provide its security do not hold in practice [MPO05, MPG05, PV17, BGG+14]. Hardware-based solutions suffer from glitches and transitions effects which can be a source of leakages and extensive research has been dedicated to design strategies and solutions to tackle these leakages. Software-based countermeasures suffer from an even larger security-gap which stems from unintended interactions between values in the CPU. While the origins of some micro-architectural leakages have been analyzed, its analysis remains difficult due to the closed-source nature of many commercial processors. A typical leakage in software implementations arises from a transition of values in a register (or a memory cell) within the micro-controller which may be unknown to the developer. On hardware platforms, an elegant masking countermeasure called *threshold implementations* proposed by Nikova *et al.* [NRR06] enabled first-order secure maskings even in the presence of glitches. However, securing maskings on a micro-architecture platform is not solved by this approach alone .

The difficulty of protecting a software masked implementation resulted in various approaches to evaluate the security of a masking scheme. Leakage simulators such as ELMO [MOW17] or MAPS [CGD18] aim to provide an easy-to-use tool for the evaluation

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

of a given assembly implementation while tools such as ROSITA [SSB+21] can perform automatic algorithmic corrections based on the leakages simulated by ELMO. However, these tools (and their resulting corrected algorithms) come with the limitation of being as good as the leakage model from which the emulator is constructed and tailored for a specific core (*e.g.* ARM Cortex) and can of course not provide formal proofs. Instead, formal verification tools such as MaskVerif [BBFG18], REBECCA [BGI+18], or SILVER [KSM20] have mostly been applied to hardware implementations although more recently the scVerif [BGG+21] tool allows for a more precise verification by capturing detailed leakages from a target using domain specific language. Gigerl *et al.* [GHP+21] bring together formal and empirical verification by analyzing the security of a software masked scheme executed on a CPU from its netlist and proposed a collection of (mostly hardware) modifications to apply on the design of a RISC-V core.

While threshold implementations were originally designed as a countermeasure for masked hardware implementations, Sasdrich *et al.* [SBM18] studied the efficiency of such masking methodologies on a software environment. Their experiment showed a first-order secure implementation of the PRESENT cipher indicating that some properties of threshold implementations may also benefit software masking schemes.

Contributions. In this work, we provide masking properties which are necessary to algorithmically secure maskings, meaning that the properties can be achieved without hardware modifications, in scalar cores with the goal of first-order probing security including glitch and transition effects. More specifically, this paper provides the following points of contribution.

We go through the known literature and summarize a list of leakages due to glitch and transition effects in various components of micro-architectures. In particular, the discussed components are standard among scalar micro-processors.

From the previous list of leakages, we propose masking notions, based on the placement of the shares, which are necessary to secure against them. We extend the notions of threshold implementations and propose stricter versions of non-completeness and uniformity which are adapted to masked micro-architectures.

We discuss how to secure maskings using notions extended from threshold implementations with $td+1$ and $d+1$ shares (where $d=1$ the security order and t the degree of the function), and provide a secure software masking of a Toffoli gate using the minimal number of shares. The security of this operation is formally proven in the glitch and transition extended probing model.

Finally, we secure a two-shared PRESENT cipher, two `Keccak` variants (*i.e.* `Keccak-`f[800] and the standard `Keccak-`f[1600]), and the 4-bit quadratic classes with the extended threshold implementation notions using no additional randomness during the computation. To show the soundness and portability of our methodology, these primitives are implemented on a RISC-V and an ARM Cortex-M4 core where we show first-order resistance over practical measurements up to one million traces. We emphasize the importance of our proposed notions by showing leakage results when the placement and shifting of the shares is done in a naive way.

2 Preliminaries

In this section, we introduce Boolean masking and threshold implementations as well as the side-channel security model considered in this paper, namely the probing model.

2.1 Boolean Masking

Boolean masking is a sound and widely-deployed countermeasure against side-channel analysis which was first introduced independently by Chari *et al.* [CJRR99] and Goubin-Patarin [GP99]. As a means to conceal a key-dependent variable $x \in \mathbb{F}_2$, a d^{th}-order Boolean masking splits such a variable into $d+1$ shares $\bar{x} = (x_0, x_1, \ldots, x_d)$ where shares $x_1, \ldots, x_d$ are drawn from a uniform distribution and x_0 is computed such that $x = \bigoplus_{i=0}^{d} x_i$. More specifically, such a masking is called a *uniform masking*.

Definition 1 (Uniform masking)**.** A random masking $\bar{X}$ over $\mathbb{F}_2^{ns}$ in s shares is uniform if for all $i \in \{0, .., s-1\}$

$$P((X_0, ..., X_{i-1}, X_{i+1}, ..., X_{s-1}) = (x_0, ..., x_{i-1}, x_{i+1}, ..., x_{s-1})) = 2^{-n(s-1)}\,.$$

In words, if every set of $s-1$ shares act as a set of uniform random variables.

2.2 Threshold Implementations

In 2006, Nikova *et al.* [NRR06] introduced a specific case of Boolean masking called a *threshold implementation* which secures the masking in the presence of glitches in hardware circuits. In its essence, a threshold implementation takes as input a uniform masking $\bar{x} = (x_0, \ldots, x_{s_x-1})$ of a secret value x with a masking $\bar{F}(\bar{x}) = \bar{y} = (y_0, \ldots, y_{s_y-1})$ of a function $F(x) = y$ such that each coordinate function f_i of $\bar{F}$ takes shares of x and produces a share y_i as output. In the following, we recall the properties of threshold implementations.

Definition 2 (Correctness [NRR06])**.** The masking $\bar{F}(\bar{x})$ is a correct masking of F when $\sum_i f_i(\bar{x}) = F(\sum_i x_i)$.

Glitches in a hardware implementation can cause unexpected leakage between the shares of a secret variable, hence reducing the security of the Boolean masking scheme (further explained in Section 2.3). In order to prevent this effect, a threshold implementation makes use of non-complete coordinate functions.

Definition 3 (Non-completeness [NRR06])**.** A function $\bar{F}(\bar{x})$ is non-complete if each of its coordinate functions f_i uses at most $s_x - 1$ input shares of $\bar{x}$.

A typical threshold implementation will consist of a collection of Boolean functions where the outputs of one masked function will be used as inputs in another one. To ensure each function is given a uniform input masking as per Definition 1, we require that a masked function outputs a uniform output masking.

Definition 4 (Uniformity [NRR06])**.** A masked function $\bar{F}(\bar{x}) = \bar{y}$ is uniform if $\forall x \in \mathbb{F}$, $\forall \bar{y} \in Sh(F(x))$:

$$\left|\left\{\bar{x} \in Sh(x) \,\middle|\, \bar{F}(\bar{x}) = \bar{y}\right\}\right| = \frac{|\mathbb{F}|^{s_x-1}}{|\mathbb{F}|^{s_y-1}}\,,$$

where $Sh(x)$ denotes the set of valid share vectors $\bar{x}$ of the secret x.

2.3 The Probing Model

In the probing model introduced by Ishai, Sahai, and Wagner [ISW03], an adversary $\mathcal{A}$ is allowed to observe a set of at most t (predefined) wires of a circuit at each execution of the masking. The security of a given implementation is proven by showing that a simulator $\mathcal{S}$ can perfectly simulate any set of at most t probes without any knowledge of the input shares $(x_0, \ldots, x_{n-1})$. A circuit ensuring this condition for any set of size t is said to

be t-probing secure. Dhooghe *et al.* [DNR19] showed that a threshold implementation achieves first-order probing security.

While the probing model is a helpful tool to formalize the security of a circuit, its model remains abstract and lacks the incorporation of physical defaults such as transition-based leakages or glitches which can alter the security order of a masking scheme. In the work by Faust *et al.* [FGP+17], a new model is introduced called the *robust probing model* which extends the probing model in order to capture physical defaults of a circuit. As a result, specific models are defined which enable an adversary $\mathcal{A}$ to use $\epsilon-$extended probes revealing additional information from a probed wire.

The first effect introduced by Faust *et al.*, which is relevant to this work, pertains to glitches. In a circuit, combinatorial logic cells are connected to each other such that the input of one cell is the output of another. During each cycle of execution, the circuit will be evaluated and the result of cells will be updated until the last cell stores its result into a register. Because of different wire length, wire speed, or gate propagation time, the result of each cell might change multiple times before the overall signal stabilizes. Regarding masked implementation, glitches pose a serious risk of information leakage on a secret value. Following the work by Faust *et al.*, the probing model is extended to capture this effect as follows.

"**Specific model for glitches.** For any ϵ-input circuit gadget `G`, combinatorial recombinations (aka glitches) can be modeled with specifically ϵ-extended probes so that probing any output of the gadget allows the adversary to observe all its ϵ inputs."

The second effect relevant to this work is on transition leakage. This effect originates from the fact that the power consumption of a CMOS circuit is dominated by its dynamic power consumption. When a value stored in a memory cell is overwritten with a different one, an adversary can measure a peak in the power consumption which can be modeled as the Hamming distance between the two values. The probing model is extended in the work by Faust *et al.* to capture this effect as follows.

"**Specific model for transitions.** For a memory cell m, memory recombinations (aka transitions) can be modeled with specifically 2-extended probes so that probing m allows the adversary to observe any pair of values stored in 2 of its consecutive invocations."

3 Micro-architectural Leakage Sources

Central Processing Units (CPUs) and Micro-Controller Units (MCUs) follow a multi-pipeline design which splits data execution into three main stages (see Figure 1) namely: the fetch stage, the decode stage, and the execute stage. The Instruction Fetch (IF) stage retrieves the instruction from memory held by the Program Counter (PC) and supplies it to the Instruction Decode (ID) stage. The latter stage interprets the fetched instruction and passes it to the EXecution stage (EX) which will forward the operation to the appropriate functional component. Each stage internally relies on specific hardware units in order to carry out a stage's tasks.

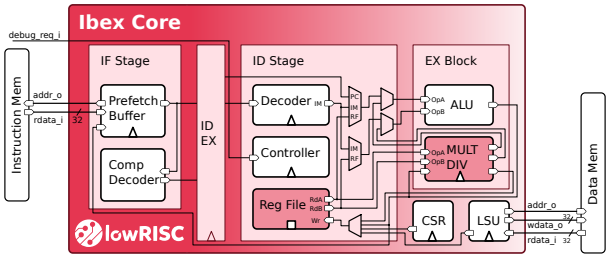


Figure 1: Ibex RISC-V Core [tea].

Even though instruction set architectures are well defined, processors embed a collection of undocumented micro-architectural features which generate unexpected recombinations of data. It should be noted that the leakage sources differ from core to core, as each target will have a specific implementation — hence different behavior — which might change the way data is internally handled. However, aside some variations, CPUs commonly mask sensitive components that produce unintended recombination of shares. In this section, we aim to provide a classification of the three main types of leakage impacting masked software implementations on low to mid range scalar cores such as ARM Cortex-M devices or RISC-V scalar cores. This classification is mainly based on the results of previous works. For each of these types of leakage namely: transition, glitch, and bitwise interaction related leakages, we detail the CPU components from which these leakages stem. It is important to note that we consider the effects of glitches and transitions throughout the micro-architecture separately, meaning that we ignore the potential combined effects of glitches and transition effects as detailed by Faust *et al.* [FGP+17].

3.1 Transition Leakage

The first effect studied on the level of micro-architectures are transition leakages as explained in Section 2.3. Multiple components of the CPU are subject to transition leakage. We go over these components.

Register File The register file holds the general purpose registers of the micro-processor which consists of n m-bit registers (*e.g.* 31 32-bit registers in the Ibex core) available to read and write a given number of values within one clock cycle (*e.g.* the Ibex core has two read ports and one write port). In a software masked implementation, this unit will hold shares of secret values used in the current computation which makes it a sensitive component of the CPU from which transition leakage can occur [BGG+14, PV17, SSB+21]. Hence, in the case of masked implementations this can be a significant issue if the old value of a register (*e.g.* $x \oplus m$) is overwritten by a new one using the same mask (*e.g.* m) as such operation will leak secret information (*e.g.* $HD(x \oplus m, m) = HW(x)$).

Pipeline Registers The Arithmetic Logic Unit (ALU) is the combinatorial block which implements bitwise operations and integer arithmetic. It operates in the execution stage and receives its instruction's operands from the decode stage. In order to propagate data through the different stages, CPUs use specific registers known as "pipeline registers" serving as intermediate registers holding data from one stage to the other. The update of the values held in these registers in case of sequential instructions might generate unexpected transition-based leakages between the operands of the instructions [SSB+21, MPW22, CGD18, PV17]. For example, assume an initial ALU instruction such as `add rA rB` is executed (where rX defines a register which contains a value X). After the decode stage, operand rA might be stored in pipeline register `P0` while operand rB is stored in pipeline register `P1`. A consecutive instruction such as `add rC rD` will update register `P0` and `P1` leading to leakage of $HD(rA, rC)$ and $HD(rB, rD)$. Note that the storage of an instruction's operands in these internal registers will vary from core to core as detailed in [GOP21]. Such micro-architectural components can be a source of unexpected leakage. As recently observed by Gao *et al.* [GOP21] and Marshall *et al.* [MPW22], instructions will not always follow the same distribution of operands in the pipeline registers: some instructions may overwrite the first pipeline register with an operand (*i.e.* the first) while some do not.

It is assumed that in an n-stage pipeline scalar core, there are n instructions in flight per cycle (one in each stage). Control-flow instructions create changes at different stages. For example, a conditional branch can occur during the decode or execute stage while an

unconditional branch can happen during the decode or fetch stage. Hence, while a given instruction might never actually be executed due to control-flow changes, its operands may already reside within some pipeline registers resulting in their leakage as observed by Marshall *et al.* [MPW22]. From the same work, we learn that while leakage between operands of instructions separated by control-flow is expected in cores using branch delay slots optimization, recombinations also unexpectedly appear in some ARM cores.

Load-store Unit The Load–Store Unit (LSU) operates during the execution stage and handles instructions between memory and the registers. The LSU embeds a storage element that stores the most recent value stored or loaded from the memory [PV17, MPW22, SSB+21]. Hence, when loading from or storing to memory, the value of this storage element is overwritten, leaking the Hamming distance between the previous and the new value. As observed in [MPW22, GHP+21, SSB+21], the load (resp. store) operation on a byte or halfword will fill the memory bus by the whole word that contains the wanted byte (or half word) which will generate unintended memory interaction potentially leading to recombination of shares.

Data Memory Similar to a register overwrite, writing data to memory interacts with data already stored in the same location [SSB+21, MPW22]. Hence, overwriting one masked value with another may remove the mask.

3.2 Glitch Related Leakage

The second effect we consider is a glitch as explained in Section 2.3. We go over the components on a software platform where glitches can cause harmful leakage.

Register File Address Decode Logic While software masked implementations were often thought to be free of glitches due to their software nature, the design of the register file is however susceptible to glitches leading to unexpected recombinations of shares [GHP+21, PV17]. Figure 2 illustrates a generic design of a register file where a 5-bit address signal connected to a multiplexer tree of depth five controls the selection of registers to be read or written during an instruction. As the address signals result from combinatorial logic performed at the same time as a given read (or write) instruction, they are subject to glitches which lead to potential sequential access to multiple registers within one clock cycle until the signals stabilize which —in software masked implementation— may result in shares recombination. Moreover, a simple bit-value change from the address signal will show transition from one register to another on the wire connecting two layers of the multiplexer tree. For example, sequential access to register `x1` and to `x4` will switch the value of `Addr[5]` from 0 to 1 leading to a transition (*e.g.* $HD(x_1, x_2)$) of value wire `L0` of the first multiplexer. Finally, the micro-controller might perform unintended reads from registers due to its interpretation of instruction bits at specific indices as operand addresses (*e.g.* "`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."[GHP+21]).

Load-store Unit In addition to transition-based leakages during a load/store byte/word instruction, a multiplexer is used in order to output the wanted byte (or halfword) and —due to possible glitches in the multiplexer selector— bytes (or halfwords) within a word may also interact unintentionally [GPM21].

Data Memory Data memory suffers similar glitch issues as for the register file [GHP+21]. Hence, storing shares within data memory without special care might lead to unintended access.

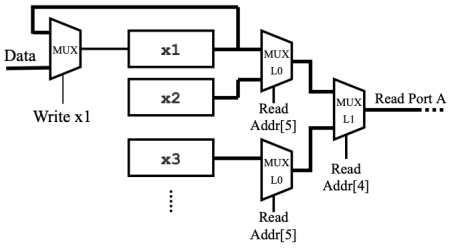


Figure 2: Register file [GHP+21]

3.3 Bitwise Interaction Leakage

In previous works, the assumption was made that the values in a register leak separately. This allowed designers to store all shares of a secret in a single register. This assumption is called the "bitwise independent assumption" and is formally given in the work by Gao *et al.* [GMPO20].

In this section, we go over the leakage effect where the bits in a single register get combined. In other words, an effect which violates the bitwise independent assumption. This leakage effect is in its essence not different from the effects of the previous section, it originates from the architecture of an ALU (the parallel execution of many operations where only one operation is chosen as the output) and from potential glitch effects which occur in this component.

Always-active Computation Units Computation units within the ALU such as bitwise operations (such as SUB, AND, ADD, OR, XOR, SHIFT, ...) are always active during the execute state. The results produced are given as input to a multiplexer which will select the appropriate result of the current instruction [GHP+21]. While bitwise operations such as AND or XOR only operate on individual bits, ADD or SHIFT will create interactions between the bits within one operand violating the bit-independence assumption required in some masking schemes storing all the shares into one register [BDF+17]. Hence, when using such a masking scheme, the execution of a simple bitwise instruction leaks. For example an always active barrel shifter can undermine share-slicing techniques as described in the work by Gao *et al.* [GMPO20].

4 Adapting Threshold Implementations for Software

Having described a list of leakage effects in Section 3, we provide necessary properties how to secure a masking against them. More specifically, we extend the non-completeness notion from threshold implementations as defined in Section 2.2. In Section 5, we back up our new notions with a two-shared masked `Keccak` whose security is verified in practice.

4.1 Register Non-completeness

Given the effects listed in Section 3, extended notions of non-completeness are required to secure maskings. For that purpose, we consider *operations* as Boolean functions which take registers as input and provide a register as output. We refer to a bit in a register at "index" i as the i^{th} bit in that register.

From Section 3.3, we find that the ALU recombines bits within a single operand (register). As a result, we cannot place all shares of a variable in a single register. The notion of non-completeness therefore needs to necessarily span all values in a single register.

Definition 5 (Horizontal non-completeness)**.** A masked operation with input registers $R0, ..., R\ell$ is *horizontal non-complete* when the set of all values in $R0, ..., R\ell$ do not contain all shares of a variable.

To achieve the above notion, it is necessary that each register in the register file contains a non-complete set of shares. In the case of a two-shared implementation, we thus enforce that each registers only holds at most one share per secret value.

From Section 3.2, we find that glitches in the address decode logic of the register file (or memory unit) can cause recombinations of values between registers. Since the multiplexer trees decode every bit of the registers separately, such recombinations only happen at the same index between registers. Similarly, due to the effect of transitions as described in Section 3.1, values at a fixed index between separate registers can leak due to them being overwritten. Due to the glitches in the address decode logic, all values in registers on a specific index have dependent leakage. As a result, we cannot place all shares of a variable on the same index over separate registers. The notion of non-completeness therefore needs to necessarily span across the separate registers at a fixed index.

Definition 6 (Vertical non-completeness)**.** A masked operation computing on m-bit registers $R0, ..., R\ell$ is *vertical non-complete* when the set of all values at index $0 \leq i \leq m$ of $R0, ..., R\ell$ do not contain all shares of a variable.

Again, at minimum the above notion also needs to hold for the memory and register file. Therefore, given an n m-bit register file, all values at index $0 \leq i \leq m$ of the n registers are non-complete. In the case of a two-shared implementation, we thus enforce that for any set of registers, the set of values at index $0 \leq i \leq m$ holds at most one share per secret value. Similarly, every separate register can hold at most one share per secret value.

We can also extend the notion of a uniform masking from Definition 1. Whereas the notions of horizontal and vertical non-completeness were necessary for security, the following notion on uniformity is neither necessary nor sufficient. However, the notion allows for a secure sequential composition of functions similar to the regular notion of uniformity.

Definition 7 (Register uniform masking)**.** Given a masking in s shares, a set of $\ell + 1$ registers $R0, ..., R\ell$ are a *register uniform masking* when for every set of $s - 1$ indices the set of shares at those indices jointly act as uniform random variables. Similarly, for every set of $s - 1$ registers, the shares in those registers jointly act as uniform random variables.

A *register uniform function* is then one which maps a register uniform input masking to a register uniform output masking.

The above definition of uniformity essentially requires the regular notion of uniformity and for the different shares to be vertically separated. Take for example a uniform three-shared $\bar{x} = (x_0, x_1, x_2)$, if we would store the three shares in three three-bit registers $R0 = [x_0, 0, 0]$, $R1 = [x_1, 0, 0]$, and $R2 = [0, x_2, 0]$ then the masking would still be uniform (from Definition 1) and horizontal and vertical non-complete but it would not be register uniform (since it essentially compressed a three-sharing to a two-sharing). Instead, we need to store the three shares in three three-bit registers such that $R0 = [x_0, 0, 0], R1 = [0, x_1, 0]$, and $R2 = [0, 0, x_2]$. Similarly, the shares also need to be horizontally separated. Coming back to the previous example, storing shares in two registers such as $R0 = [x_0, x_1, 0]$ and $R1 = [0, 0, x_2]$ would also not be register uniform.

As opposed to hardware operations, operations on micro-controllers are often done sequentially. As a result, we cannot make an operation which is correct, register non-complete, and register uniform in a single instruction. Instead, we will require the composition of several operations (implementing a masked function) to be register uniform and its intermediate stages to be register non-complete or, more strictly, robust probing secure considering the effects from Section 3.

Together, the two notions of register non-completeness and register uniformity protect micro-processors against the effects listed in Section 3. However, in practice, due to each micro-processor being of a different (and unknown) design, we often cannot ensure register uniformity as values might be stored in unaccounted registers and kept throughout longer periods of the computation. Nevertheless, the above notions can help a designer to attain some necessary properties for a secure masking. Independently to this work, a public repository on Github containing a software masking of ASCON [Sch] also rotates the shares against each other to reduce leakage. In the repository, two and three shares masked ASCON software implementations as well as TVLA results are available. While these ad-hoc implementations show interesting results for the three-shared implementations, the two-shared variants require device specific fixes in order to ensure practical security. Indeed, as opposed to our work, shares are left unrotated during specific operations (*e.g.* non-linear operations). We emphasize that keeping such a rotation present throughout the nonlinear operations can secure the implementation without additional measures specific to the platform.

4.2 Register Non-completeness with $td + 1$ Shares

We start with the traditional setting from threshold implementations where we have $td + 1$ shares to protect a function of degree t against d^{th}-order probing adversaries. Since we focus only on first-order protection, we take $d = 1$. In order to achieve a masking with registers which is horizontal non-complete (Def. 5), we separate the registers in $t + 1$ domains. Each share is assigned to a single domain. In order to fulfill vertical non-completeness (Definition 6), we require that no vertical alignments of related shares between any of the domains occur during the computation. To this end, we store shares in the registers shifted by a unique value depending on which domain they belong to. An example is shown in Figure 3 where the input (or output) is separated by share and shifted.

Given an arbitrary masked function $\bar{F} : \mathbb{F}_2^{n(t+1)} \to \mathbb{F}_2^{n(t+1)} : \bar{x} \mapsto \bar{F}(\bar{x})$ which is non-complete and uniform given the traditional notions from Section 2.2, the function can be made register non-complete and register uniform by dividing $\bar{x}$ in $t + 1$ registers each holding one share and shifting each domain such that they are not vertically aligned. As a general methodology, this masked function can be computed by shifting the needed input shares underneath each other and calculating the respective output share. The input shares are then shifted back into place and the process is repeated until all output shares are calculated. Once the calculation is complete, we remove or overwrite the inputs shares $\bar{x}$ as the input concatenated with the output is not register uniform. For example, for three shares and the non-complete and uniform masked function $\bar{F}$, the input (*e.g.* x_1) is rotated to the left so that the first output $F_0(x_0, x_1)$ share can be calculated. Afterwards, x_1 is rotated back at its original position and the process is repeated to get all the outputs. Finally, the inputs are then cleared for the state to be register uniform (see Figure 3).

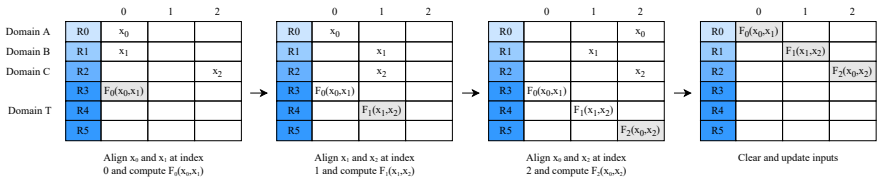


Figure 3: Methodology for $td + 1$ shares. In this masking, registers R0, R1, R2 belong to domain A, B and C respectively and R3-R5 are temporary registers in domain T.

Finding a methodology or maskings of specific functions which do not require the clearing of input variables or which require fewer temporary registers is left as an open

problem.

4.3 Register Non-completeness with $d + 1$ Shares

Working with fewer shares makes the extended notions of non-completeness less trivial to achieve. In order to satisfy Definition 5, we again separate registers into several domains. In addition, we are required to use a "temporary domain" (called domain T) which can hold any cross-products which do not belong to the main domains. A depiction of the layout is given in Figure 4. Similar to the $td + 1$ shared case, each domain (including the temporary domain) is again shifted to ensure vertical non-completeness (Definition 6). For example, for two-shared implementations, shares held within domain A will all be stored at index 0 while the shares in domain B will all be stored at index 1.

When performing nonlinear operations, cross products (*e.g.* $a_i b_j$) will be created which —if stored in the same register holing either a_i or b_j— will cause harmful leakage due to the effect listed in Section 3.3 on bitwise interactions. Instead, domain T is allocated to hold these cross products until the products are recombined such that the result can be stored in the main domains again, in which case the values in the temporary register can be cleared or overwritten by independent data. Similarly, to avoid harmful leakage from the effect listed in Section 3.2 on glitches in the decode logic, the cross products in domain T are stored at separate indices (*e.g.* for two-shared implementations $a_0 b_1$ is stored at index 2 and $a_1 b_0$ at index 3. Note that cross products $a_0 b_0$ and $a_1 b_1$ can be held at the same index as the first and second share, respectively).

It is possible to find a register non-complete masking of an arbitrary function. Given a function $F : \mathbb{F}_2^n \to \mathbb{F}_2^n$ consisting of u variables, from the original paper on threshold implementations [NRR06, Corrolary 1] we find that there exists a non-complete masking of F with $1 + 2^{nu}$ output shares. By allocating a total of $1 + 2^{nu}$ temporary registers and storing each output share in a separate register and index, the output is horizontal and vertical non-complete.

Finding a masking which achieves register non-completeness *and* uniformity is more complex. However, the $d + 1$ non-complete and uniform maskings provided in the work by Shahmirzadi and Moradi [SM21] can easily be transformed to be register non-complete and uniform by storing each cross-product in a separate register and index.

For example, take the masking of the AND gate from [SM21]. The required inputs are rotated back and forth to compute each cross product in a separate temporary register at a unique index. The inputs are cleared from the register file in order to enable the construction of the compressed and uniform outputs from the cross product components which are then rotated to the regular indices (see Figure 4).

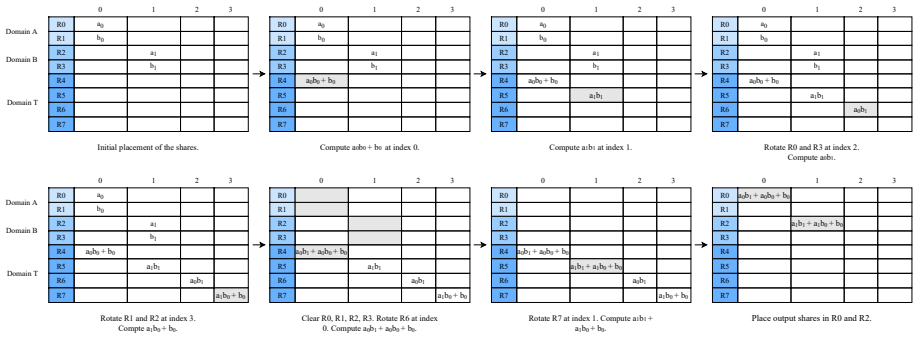


Figure 4: Methodology for $d + 1$ shares. In this masking, registers R0-R1, R2-R3 belong to domain A and B respectively and R4-R7 are temporary registers in domain T.

$$\begin{array}{llll} f_0(a_0, b_0) & = a_0 b_0 & \to x'_0 & \\ f_1(a_0, b_1, r) & = a_0 b_1 + r & \to x'_1 & x'_0 + x'_1 = x_0 \\ \hline f_2(a_1, b_0, r) & = a_1 b_0 + r & \to x'_2 & x'_2 + x'_3 = x_1 \\ f_3(a_1, b_1) & = a_1 b_1 & \to x'_3 & \end{array},$$

Investigating maskings of functions which achieve register non-completeness and uniformity using minimal temporary registers is left as future work.

A Secure Toffoli Gate To illustrate register non-completeness in $d + 1$ shares, we detail the computation of a Toffoli gate using two shares, namely the function which maps the three bits (a, b, c) to $(a, b, c + ab)$. The masking of this function can be securely achieved using only one temporary register.

In domain A, registers $R0$ to $R2$ will store a_0, b_0, c_0 at index 0 while registers $R3$ to $R5$ of domain B will hold the second shares a_1, b_1, c_1 at index 1. Quadratic terms using shares of similar domain fulfill vertical non-completeness (Definition 6) and can be stored securely in a temporary register in domain T under the same index as their domain (*e.g.* $a_i b_i$ can be placed in a temporary register at index i). However, care must be taken regarding the cross-domain quadratic terms of the function. Since the computation of $a_0 b_1$ combines shares of the domains A and B, the result has to be stored in a temporary register at an index i where $i > 1$. Practically, in order to compute such a term, the two registers holding the shares will have to be aligned (*e.g.* shifted) at the same index position i. Figure 5 illustrates the eight main steps required for the construction of the outputs c_0 and c_1. In Table 1 we provide a step by step execution of this Toffoli gate on 1-bit size shares which requires 1 temporary register and 21 cycles (compared to 9 cycles in the unshifted variant). Note that in such example, we enforce that any shifted register of domain A or B to be shifted back at their original position which counts for 4 additional cycles. In addition, for the operation to be register uniform (Definition 7) we need to clear the temporary register (R6) at the end of the operation. However, we note that repeating the same operation would securely overwrite R6 by the cross product $a_0 b_0$ at index 0 thus this clearing operation is not always needed when optimization is required. Note that storing in memory the updated results of the Toffoli gate would still be insecure due to transition leakage from the original and updated value in memory (*e.g.* $\mathrm{HD}(c_0 + a_0 b_0 + a_0 b_1, c_0)$).

This example is made with a CPU using a simple ISA (such as `RV32I`). However, CPUs such as Cortex-M3 which support a larger instruction set such as the Thumb-2 might benefit from "free" shift operation which would reduce the total cycle count. In addition, by working over 32-bit words such as in the `Keccak-`f[800] masking in Section 5, we can simultaneously compute 32 of such Toffoli gates.

Robust Probing Security We show that the Toffoli gate is correct, register uniform, and glitch and transition-extended probing secure considering the leakage effects from Section 3. In other words, we consider a probing adversary who can read all values in a register or a value at a fixed index from all registers in the register file.

Consider the output of the Toffoli gate $(a_0, a_1, b_0, b_1, c_0 + a_0 b_0 + a_0 b_1, c_1 + a_1 b_1 + a_1 b_0)$, it is clear that $a_0 + a_1 = a$, $b_0 + b_1 = b$, and that $c_0 + c_1 = c + ab$. Thus, the computation is correct. Additionally, since the masking can be seen as a Feistel operation adding computations on (a_0, a_1, b_0, b_1) to the shares (c_0, c_1), it is clear that the output is also uniform (following Definition 1) since it is invertible. Since the output is again separated into the same domains and correctly shifted, the output masking is register uniform (following Definition 7).

It is sufficient to show that the computation is probing secure including the extensions from Section 3. This means we have to show that a simulator can simulate the probed

Table 1: Toffoli gate executed in 21 cycles using one temporary register.

#	Operation	Reg.	Index	#	Operation	Reg.	Index
0	a_0b_0	$R6$	0	11	$a_1b_1 + c_1$	$R5$	1
1	$c_0 + a_0b_0$	$R2$	0	12	shift $R3$	$R3$	2
2	shift $R0$	$R0$	2	13	shift $R1$	$R1$	2
3	shift $R4$	$R4$	2	14	a_1b_0	$R6$	2
4	a_0b_1	$R6$	2	15	shift back $R3$	$R3$	1
5	shift back $R0$	$R0$	0	16	shift back $R1$	$R1$	0
6	shift back $R4$	$R4$	1	17	shift $R5$	$R5$	2
7	shift $R2$	$R2$	2	18	$c_1 + a_1b_1 + a_1b_0$	$R5$	2
8	$c_0 + a_0b_0 + a_0b_1$	$R2$	2	19	shift back $R5$	$R5$	1
9	shift back $R2$	$R2$	0	20	clear $R6$	$R6$	2
10	a_1b_1	$R6$	1				

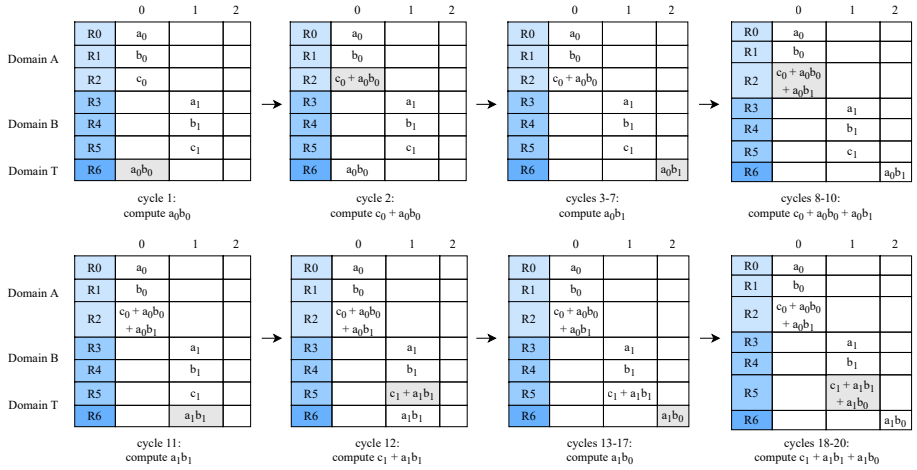


Figure 5: Outputs c_0 and c_1 of the Toffoli gate. In this masking, registers R0-R2 belong to domain A, R3-R5 belong to domain B, and R6 is a temporary register in domain T.

result from scratch. In other words, that the probed values behave randomly.

First, note that the AND-XOR operation requires only three indices (bits) of seven registers. In the following table, we denote the shares in the first, second, and third position on these registers.

Second, note that each register only holds a single value and that each double-input operation always calculates on values which are vertically aligned. As a result, the leakage from Section 3.3 on bitwise interaction is already captured by the leakage from Section 3.2 on glitches in the address decode logic. In other words, a glitch-extended probe can only see one cell of Table 2. Due to transition leakage, this probe will also view the shares from the same position in the cell above.

We now claim that there is a simulator which is capable of simulating the probed values in the operation from scratch. From Table 2, we can categorize the following types of probed information.

1. A probe views subsets of a_0, b_0, c_0 or a_1, b_1, c_1.
2. A probe views subsets of $c_0 + a_0b_0, a_0, b_1$ or $c_1 + a_1b_1, a_1, b_0$.
3. A probe views subsets of $a_0, b_0, c_0 + a_0b_0 + a_0b_1$ or $a_1, b_1, c_1 + a_1b_1 + a_1b_0$.

We define our simulator as follows.

- For a probe in the first category, the simulator either samples a_0, b_0, c_0 or a_1, b_1, c_1 as random values.

Table 2: Probed variables per cycle of the Toffoli gate masking.

#	Position 0	Position 1	Position 2
0	a_0, b_0, c_0	a_1, b_1, c_1	
1	a_0, b_0, c_0, a_0b_0	a_1, b_1, c_1	
2	$b_0, c_0 + a_0b_0, a_0b_0$	a_1, b_1, c_1	a_0
3	$b_0, c_0 + a_0b_0, a_0b_0$	a_1, c_1	a_0, b_1
4	$b_0, c_0 + a_0b_0$	a_1, c_1	a_0, b_1, a_0b_1
5	$a_0, b_0, c_0 + a_0b_0$	a_1, c_1	b_1, a_0b_1
6	$a_0, b_0, c_0 + a_0b_0$	a_1, b_1, c_1	a_0b_1
7	a_0, b_0	a_1, b_1, c_1	$c_0 + a_0b_0, a_0b_1$
8	a_0, b_0	a_1, b_1, c_1	$c_0 + a_0b_0 + a_0b_1, a_0b_1$
9	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	a_1, b_1, c_1	a_0b_1
10	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	a_1, b_1, c_1, a_1b_1	a_0b_1
11	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	$a_1, b_1, c_1 + a_1b_1, a_1b_1$	
12	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	$b_1, c_1 + a_1b_1, a_1b_1$	a_1
13	$a_0, c_0 + a_0b_0 + a_0b_1$	$b_1, c_1 + a_1b_1, a_1b_1$	a_1, b_0
14	$a_0, c_0 + a_0b_0 + a_0b_1$	$b_1, c_1 + a_1b_1$	a_1, b_0, a_1b_0
15	$a_0, c_0 + a_0b_0 + a_0b_1$	$a_1, b_1, c_1 + a_1b_1$	b_0, a_1b_0
16	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	$a_1, b_1, c_1 + a_1b_1$	a_1b_0
17	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	a_1, b_1	$c_1 + a_1b_1, a_1b_0$
18	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	a_1, b_1	$c_1 + a_1b_1 + a_1b_0, a_1b_0$
19	$a_0, b_0, c_0 + a_0b_0 + a_0b_1$	$a_1, b_1, c_1 + a_1b_1 + a_1b_0$	a_1b_0

- For a probe in the second category, the simulator either samples a_0, b_1, c_0 or a_1, b_0, c_1 as random values. We see that c_0 (similarly c_1) perfectly masks a_0b_0 (similarly a_1b_1).
- For a probe in the third category, the simulator either samples a_0, b_0, c_0 or a_1, b_1, c_1 as random values. We see that c_0 (similarly c_1) perfectly masks a_0b_1 (similarly a_1b_0).

Since we went over all probe position, we have shown the glitch and transition-extended probing security of the masked Toffoli gate.

5 Software Threshold Implementation of `Keccak`

In this section, we detail the properties required for the implementation of our two-shared first-order secure threshold implementation of `Keccak`. We first recall the basic structure of `Keccak`. Afterwards, we discuss its masking and motivate our approach based on the leakages discussed in Section 3.

5.1 `Keccak`

`Keccak` [BDPA13] is a family of sponge-based hash functions based on a permutation `Keccak-`$f[b]$ of state-size $b = r + c$ where the rate r defines the input block size and the capacity c determines the security level required. The state S is organized as a 3-dimensional $5 \times 5 \times w$ matrix with $w = 2^\ell$ bits where $\ell \in [0, 6]$. Hence, a single bit in the state can be accessed via (x, y, z) coordinates while a w-bit `lane` is obtain using (x, y) coordinates. A `row` is defined as a 5 bits value given by coordinates (y, z), a `column` via (x, z), a `sheet` as 5 lanes for a fixed index x and a `plane` as 5 lanes for a fixed y coordinate. The `Keccak-`f permutation consists of Nr (e.g 22 for $w = 32$) iterations of a sequence of five operations (θ, ρ, π, χ, and ι) which manipulate the state S. θ is a linear map which computes the XOR of each bit of the state with the parity of two surrounding columns. ρ and π perform respectively a rotation and permutation on the state. χ is the only non-linear operation of the permutation where a 5-bit S-box is used on the entire

state. ι adds a round constant to the first lane of the state. More information can be found in the original reference [BDPA13].

5.2 Software Threshold Implementation of Keccak-f

In this section, we describe how we secure Keccak-f[800]. We detail on the masking of the steps which require specific care, namely: the state storage, the χ step, the θ, and the ρ step. We then provide arguments on its robust probing security as per Section 2.3.

5.2.1 State Storage

In order to enforce the properties defined in Section 4.1 on the storage of the state, we follow a similar placement of the shares as in the simple Toffoli gate example. More precisely, the second shares will be right-rotated by one relative to the first shares. Figure 6 provides an illustration on the placement of the bits in the two data blocks (represented as matrices of 5×5 lanes) corresponding to the first and second shares. Since the Keccak state is initialized at zero we construct the masked default states with shifted shares of zero values.

(a) First shares. **(b)** Second shares.

Figure 6: Placement of the first (left) and second (right) shares represented as matrices of 5×5 lanes. The first 32-bit lane of each share is detailed in the lower part of the figure with their respective (shifted) bit indices. The first share is denoted by a and the second by b where we use subscript to indicate the separate bits.

5.2.2 χ Step

The χ step is the only non-linear function in Keccak-f and operates on 5 bits (*i.e.* a `row`) of the state, updating the five planes of the state. To compute the χ step, a full `plane` is loaded into the register file from memory by filling the five registers of the domain A (resp. domain B) with the first (resp. second) shares of the state. Since the computation is performed on 32-bit lanes, 32 S-boxes are computed simultaneously. We base the masking of the χ step on the masked χ given by Daemen *et al.* [DDE+20, Sect. 3.5]. In that work, the Keccak S-box is calculated as the five times sequential application of a two-shared Toffoli gate using an extra input which is recycled using a changing of the guards construction [Dae17]. More specifically, consider the "permuted AND-NOT" function $p(a, b, c) = (a + c + bc, b, c)$ and denote its masking by $\bar{p}$ such that $p_i(\bar{a}, \bar{b}, \bar{c}) = (a_i + c_i + b_i(c_0 + c_1), b_i, c_i)$ [1]. This operation is calculated in the same manner as the Toffoli gate from Section 4.3. Consider the two-shared 5-bits $(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e})$ and an

[1] We slightly adapted the masking from the work by Daemen *et al.* which originally had $p_0(\bar{a}, \bar{b}, \bar{c}) = (a_0 + c + b_0 c, b_0, c_0)$ and $p_1(\bar{a}, \bar{b}, \bar{c}) = (a_1 + b_1 c, b_1, c_1)$. This adaptation ensures improved transition leakage protection when writing back the results from the masked χ to memory overwriting the original inputs.

Table 3: Step-by-step execution of θ on domain A using four temporary registers.

#	Operation	Reg.	#	Operation	Reg.
0	Load sheet 0	$R5$-$R9$	11	Store updated sheet 2	$R5$-$R9$
1	Sum of sheet 0	$R10$	12	Load sheet 4	$R5$-$R9$
2	Load sheet 1	$R0$-$R4$	13	Add $R12$ to sheet 3	$R0$-$R4$
3	Sum of sheet 1	$R11$	14	Sum of sheet 4	$R12$
4	Load sheet 2	$R5$-$R9$	15	Add shifted $R12$ to sheet 3	$R0$-$R4$
5	Sum of sheet 2	$R12$	16	Store updated sheet 3	$R0$-$R4$
6	Add $R10$ and shifted $R12$ to sheet 1	$R0$-$R4$	17	Load sheet 0	$R0$-$R4$
7	Store updated sheet 1	$R0$-$R4$	18	Add $R12$ and shifted $R11$ to sheet 0	$R0$-$R4$
8	Load sheet 3	$R0$-$R4$	19	Store updated sheet 0	$R0$-$R4$
9	Sum of sheet 3	$R13$	20	Add $R13$ and shifted $R10$ to sheet 4	$R5$-$R9$
10	Add $R11$ and shifted $R13$ to sheet 2	$R5$-$R9$	21	Store updated sheet 4	$R5$-$R9$

extra zero-sharing $\bar{r}$, the two-shared χ is calculated as follows

$$\begin{aligned} \bar{r} &\leftarrow \bar{p}(\bar{r}, \bar{e}, \bar{a}) & \bar{a} &\leftarrow \bar{p}(\bar{a}, \bar{b}, \bar{c})\,, \\ \bar{c} &\leftarrow \bar{p}(\bar{c}, \bar{d}, \bar{e}) & \bar{e} &\leftarrow \bar{p}(\bar{e}, \bar{a}, \bar{b})\,, \\ \bar{b} &\leftarrow \bar{p}(\bar{b}, \bar{c}, \bar{d}) & \bar{d} &\leftarrow \bar{d} + \bar{r}\,, \\ \bar{r} &\leftarrow (a_1, a_1)\,. \end{aligned}$$

In the last operation, we update the sharing of zero $\bar{r}$ with the share a_1. In order to adhere to the rotations between domains A and B, the register holding r_1 is rotated.

5.2.3 θ Step

In the θ step, each sheet has to be loaded in the register file in order to compute the parity of its columns. Because of the limited number of registers available in the register file, and the number of registers required for this step, the execution is first performed solely on domain A (*i.e.* the first shares) and then on domain B (*i.e.* the second shares). Table 3 shows a step-by-step execution of the computation of θ for domain A as the same operation is performed for domain B. The example makes use of ten registers in domain A and four temporary registers in domain T.

5.2.4 ρ Step

The linear step ρ performs a rotation on each lane of the state which could —depending of the rotation value— result in a temporary realignment of shares within the register file. As a consequence, we enforce this step to be computed solely on one share at a time. While the evaluation of the θ step needs to be done separately due to the limited number of available registers, the ρ step requires this special care due to the nature of its operation (*e.g.* rotation) and its potential effect on the placement of the shares.

5.2.5 Robust Probing Security

In this section, we show that our masking adheres to the properties given in Section 4 on the extended notions of non-completeness and uniformity. Later in Section 6, a practical validation is performed via lab experiments.

First, note that since the masked functions are register uniform (since the function is uniform and its output shares are vertically separated), each step starts from a register uniform input. We argue the linear layers $(\theta, \rho, \pi, \iota)$ and the nonlinear layer (χ) have register uniform inputs and outputs, and that their computation is robust probing secure.

- **Linear layer:** The computation of the linear steps are performed share-wise (in particular, ι only works on one share) where the register file only contains one share of the state at a time. As a result, this computation automatically is robust probing secure.
- **Nonlinear layer:** The computation of the χ step is the five-times sequential application of the Toffoli gate from Section 4.3 whose register uniformity and robust probing security was already proven in Section 4.3.

Because of each masked function being register uniform, the vertical non-completeness protects against transition leakage between each masked function. However, it is possible intermediate computational steps are stored (for example in pipeline registers) invalidating register uniformity of the state. One such example is the "Data Memory" from Section 3.1 where there is potential transition leakage from results written back into memory. In particular, our masking of the χ function was adapted to ensure this writing back does not cause transition leakage. However, these cases still need manual work. The formal verification of this manual work along with verification techniques for these pipeline registers or memory write-backs are left as future work.

6 Evaluation

This section covers the first-order side-channel security analysis of various implementations namely: two `Keccak-`f variants, the PRESENT cipher, and 4-bit quadratic classes. In order to assess the soundness and portability of our methodology, we evaluate our implementations on two different micro-controllers, namely a scalar RISC-V core and an ARM Cortex-M4 core. First, we detail specifics about the assembly implementations and the measurement setup. Second, we present results of the first-order t-test evaluations on the first-round using one million measurements.

6.1 `Keccak-`f Implementations

Our implementations follow the FIPS PUB 202 Standard [NIS15] (SHA-3) with parameters $b = 800$ or $b = 1600$, $c = 512$, a delimited suffix value `0x06` and 32-bit output. In order to ensure complete control over the registers' usage, the `Keccak-`f implementations are realized in assembly. In order to optimize the algorithms' efficiency and ease implementation efforts, the rotation (ρ), permutation (π), and round (ι) constant values are pre-computed and stored in static arrays. Note that in the case of the `Keccak-`f[1600] variant, each lane is composed of 64 bits. Hence, on a 32-bit processor each lane is coded following a bit interleaving technique [BDP+11] where a 64-bit lane is represented as two 32-bit words where the first register holds the even bits while the second register contains the odd bits. Such an optimization is helpful regarding the 64-bit rotation required in the algorithm. In order to optimize such an encoding of the lanes, we enforce the shift between the two shares to be made of an even value enabling us to use the same χ assembly code as in the `Keccak-`f[800] by running it twice (once for each of the 32-bit parts). Table 4 presents a comparison between the masked and unmasked implementations on the two cores regarding execution time (in milliseconds and clock cycles), code size, and memory usage (in bytes). As masked implementations operate twice the linear steps (once for each share), the clock cycles required for such steps are doubled compared to the unmasked versions. The masked non-linear step χ requires more cycles and results in at most three times the number of clock cycles compared to the unmasked variant. These effects can be observed notably in

Table 4: Implementation results of the `Keccak` hash function and `Keccak-`f permutation. The first four rows detail `Keccak-`f[800] while the last four rows detail `Keccak-`f[1600]. Time is given in milliseconds, and ROM and RAM are given in bytes.

	Variant	Time	Cycles			ROM	RAM
		`Keccak-`f	`Keccak-`f	$[\theta, \rho, \pi, \iota]$	χ	`Keccak`	`Keccak`
f[800]	Cortex-M4 masked	2.510	60258	28754	31770	7964	2544
	Cortex-M4 unmasked	1.148	27560	14580	13172	6868	2520
	RISC-V masked	4.825	76874	31487	48084	32294	6348
	RISC-V unmasked	2.184	34615	16885	20025	29858	6328
f[1600]	Cortex-M4 masked	4.903	117620	43436	74422	13792	2660
	Cortex-M4 unmasked	2.323	55717	21997	33925	11972	2636
	RISC-V masked	9.415	150815	43448	107654	40746	6452
	RISC-V unmasked	4.937	73900	24486	49647	37690	6424

the ARM variants. Implementations rely on rotation instructions especially in the masked variant χ step. As opposed to Cortex-M4, the RISC-V instruction set does not include rotate instructions, hence its metrics are larger than the Cortex-M4.

6.2 PRESENT Implementation

PRESENT is an ultra-lightweight block cipher which operates on 64-bit blocks with a 80-bit (or 128-bit) key size. The non-linear part of the cipher is based on a cubic 4-bit S-box which can be decomposed in a collection of affine transformations over the quadratic class Q_{12}^4. Our construction of the S-box follows the one described in the work of Sasdrich *et al.* [SBM18] where the S-box is coded as

$$S = A'' \circ Q_{12}^4 \circ A \circ A''' \circ A'' \circ Q_{12}^4 \circ A,$$

with Q_{12}^4: `0123456789CDEFAB`, A: `01AB892345EFCD67`, A': `0B835ED61A924FC7`, and A''': `8FDACB9E43160752` from which we obtain the corresponding Algebraic Normal Form (ANF). In order to perform parallel S-box computation, the 64-bit block is encoded from two 32-bit registers into 4 registers holding respectively the i^{th} bit of each S-box. Since the linear steps of the algorithm are performed share-wise and the non-linear step is based on the affine composition of the 4-bit quadratic class Q_{12}^4 which can be constructed using Toffoli gates [Dae17, Table 1], the robust probing security of the implementation follows the same reasoning as in Section 5.2.5. Table 5 presents a comparison between the masked and unmasked implementations on the two cores regarding execution time (in milliseconds and clock cycles), code size, and memory usage (in bytes). The cycle count overhead of the masked versions can be explained by the fact that unmasked variants use lookup tables for the S-box and the masked variants change the state from standard to bitsliced data representation between each S-box operation.

While the current literature does not provide many secure first-order implementations on software, a case study of masked PRESENT implementations has been made by Sasdrich *et al.* [SBM18]. They compared the security and efficiency of first and second-order Boolean masked designs and a threshold implementation using three shares on an 8-bit AVR MCU. Only the threshold implementation was shown to be secure. Section 6.5 shows the secure implementation results for our two-shared PRESENT. This shows that we directly improve the work by Sasdrich *et al.* as we can reduce the number of shares. However, a more detailed comparison in efficiency is not possible due to the difference between the AVR core and our RISC-V and ARM cores. Moreover, our design uses vectorized instructions whereas the work by Sasdrich *et al.* uses a lookup table approach. Finally, they only test the designs up to 100k traces whereas our designs are tested for 1M traces.

Table 5: Implementation results of the PRESENT cipher on the Cortex-M4 and RISC-V cores. Time is given in milliseconds, ROM and RAM are given in bytes.

Variant	Time	Cycles	ROM	RAM
Cortex-M4 masked	3.631	87166	17512	2256
Cortex-M4 unmasked	0.601	14467	6964	2264
RISC-V masked	5.965	95609	43398	6068
RISC-V unmasked	0.831	13296	30370	6080

6.3 4-bit Quadratic Classes Implementation

Since linear operations can be computed share-by-share, the difficulty of constructing a secure software masked implementation lies in the non-linear part. Hence, to illustrate the generality and efficacy of our methodology, we provide implementations of the six classes of quadratic 4-bit functions [BNN+15]. As detailed in [Dae17], those classes can be implemented using two-shared masked Toffoli gates whose register uniformity and robust probing security was already proven in Section 4.3. We evaluate the security of two-shared implementations of those classes pipelined into each other namely:

$$Q^4_{300} \circ Q^4_{299} \circ Q^4_{294} \circ Q^4_{293} \circ Q^4_{12} \circ Q^4_4 \,.$$

6.4 Measurement Setup

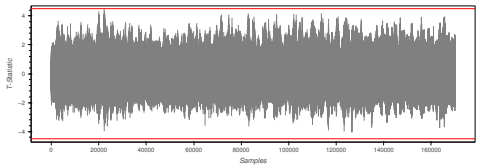
The first target platform is a FE310-G002 [SiFa] SoC which embeds a RISC-V E31 core. The E31 is a 32-bit single-issue, in-order, five-stage pipeline using the RV32IMAC instruction set architecture and an internal frequency of 16 MHz. The second target is a CW308T-STM32F target board [Newa] which embeds a STM32F415RG [STM] Cortex-M4 micro-controller using an internal 24 MHz operating frequency. The acquisition was performed using a NewAE CW308 UFO board [Newb] and a Tektronix DPO70404C oscilloscope with a sample rate of 625MS/s. An external 16 MHz and 8 MHz clock frequency was used for the RISC-V and the Cortex-M4 cores respectively. We synchronized the oscilloscope and the external clock for all our measurements. The RISC-V assembly code is compiled from the Freedom Studio IDE using the pre-built RISC-V GCC toolchain [SiFb]. The Cortex-M4 assembly implementation is compiled using the Arm GNU [2] toolchain.

6.5 Results

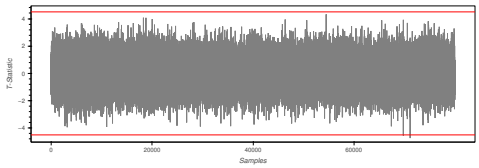
Our evaluation focuses on leakage detection using a *non-specific*, *fixed vs. random* first-order t-test statistic [GJJR11] on the computation of the first round of the primitives. We use the usual threshold value defined at $t = 4.5$ which provides a confidence of roughly 0.99999. Note that since our measurements contain a large number of sample points (*e.g.* the `Keccak-`f[800] and `Keccak-`f[1600] variants respectively contain 170k and 270k sample points), the threshold value t could be adapted to a higher value as discussed in [DZD+17, Table 1],[BGG+14, Appendix A]. The generation of the input shares as well as the zero-sharings for the randomness used in χ and for the initial state were computed externally and sent directly to the micro-controllers. Only the generation of the masked round keys in the PRESENT cipher were generated internally. We now discuss the t-test results of the `Keccak-`f[800]. Figure 7a and 7b show the t-test results where the RNG is activated. These results confirm our theoretical expectation as no significant evidence of leakage was detected for one million measurements. As a second scenario, we evaluated the permutation with the RNG deactivated (*e.g.* the random values are set to zero). As expected, the implementation leaks with only 10k traces (see Figure 7c and 7d). Last, we

[2] gcc-arm-11.2-2022.02-x86_64-arm-none-eabi

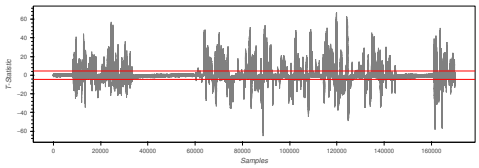
assess the impact of the shifted placement of the shares. In this scenario we evaluate the permutation with the RNG activated but with all shares aligned (*e.g.* shares are aligned at index 0 and the assembly code is stripped from any rotation operations involved in our methodology). Figure 7e and 7f show significant leakages in the non-linear step χ which stem from the leakage sources defined in section 3. As a reference, we provide the average plots of the power consumption in Figure 7g and 7h in which we can distinguish the linear steps θ, ρ, π followed by the non-linear step χ which is iterated on the five planes of the state (producing the repeated patterns on the right). Note that in the mean plots, the oscilloscope's `trigger` functions account for the few cycles displayed on the extremities of the figures. Figure 8 shows the t-test results for the PRESENT cipher, the `Keccak-`f1600 variant, and the quadratic 4-bit S-boxes on the RISC-V and the Cortex-M4 core.



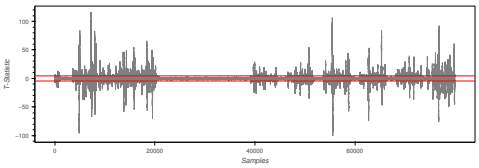
(a) RISC-V RNG activated.



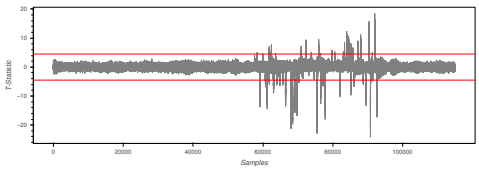
(b) Cortex-M4 RNG activated.



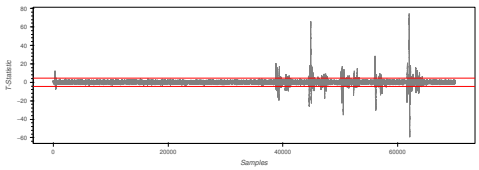
(c) RISC-V RNG deactivated.



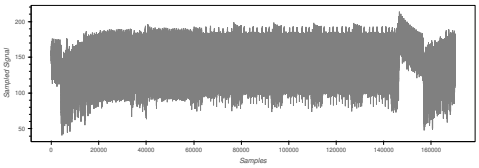
(d) Cortex-M4 RNG deactivated.



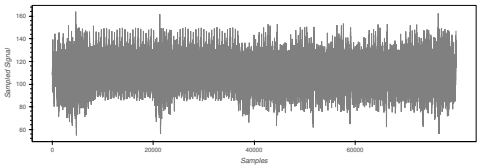
(e) RISC-V RNG activated shares aligned.



(f) Cortex-M4 RNG activated shares aligned.



(g) RISC-V Mean.

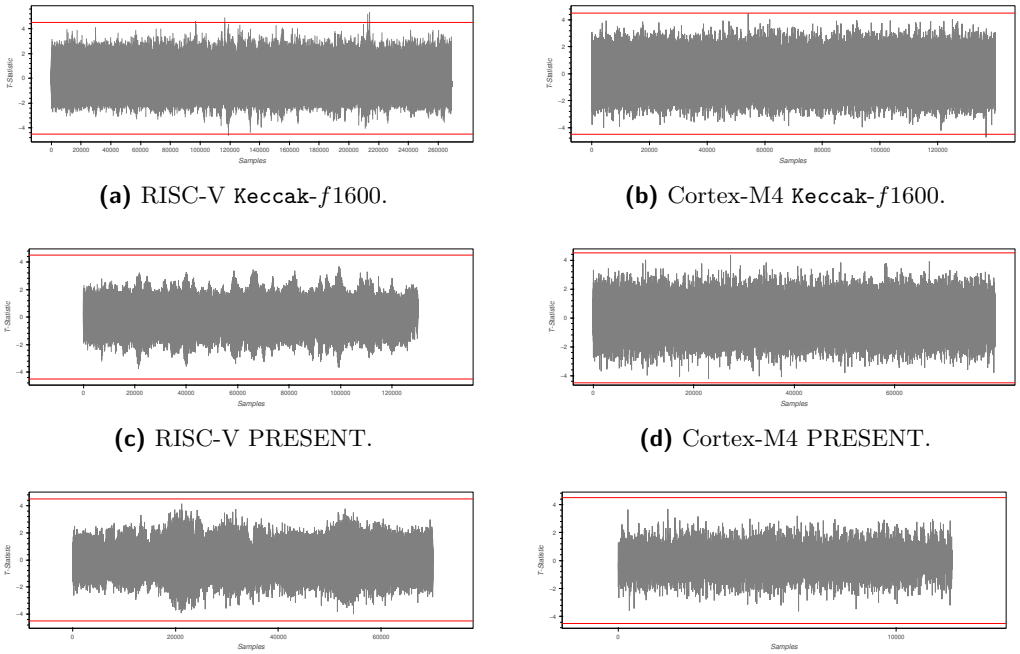


(h) Cortex-M4 Mean.

Figure 7: First-order t-test results of a `Keccak-`f[800] round. The ±4.5 threshold is marked by red lines. Experiments with RNG active use 1M traces, with RNG inactive use 10k traces, and with aligned share evaluations use 100k traces.

7 Conclusion and Future Work

We provided necessary properties for a masking in order to be secure in scalar microprocessors. These properties led to a first-order glitch and transition secure methodology based on share placement. This methodology, in turn, was applied to create two-shares



(a) RISC-V Keccak-f1600. **(b)** Cortex-M4 Keccak-f1600.

(c) RISC-V PRESENT. **(d)** Cortex-M4 PRESENT.

(e) RISC-V quadratic classes. **(f)** Cortex-M4 quadratic classes.

Figure 8: First-order *non-specific*, *fixed vs. random* t-test results on RISC-V and Cortex-M4 cores with 1M traces. The ±4.5 threshold is marked by red lines.

randomness-free Keccak-f variants, the PRESENT cipher, and implementations of the 4-bit quadratic classes. The masked primitives were implemented on a RISC-V core and on an ARM Cortex-M4 core where their security was practically validated.

In this work, we decided to focus on first-order and randomness-free maskings, hence the decision to extend the notions of threshold implementations. We believe this decision provided for a cleaner practical validation where we could not use randomness for obfuscation or noise-increasing purposes (*i.e.* no trick up your sleeve). However, we are very interested to see the same masking techniques being applied to create higher-order and composable secure maskings (*e.g.* non-interference secure) using fresh randomness. As such, we pose this line of research as interesting future work. Additionally, the aim of this paper was to show the placement of the shares builds towards algorithmic protection in micro-processors. We did not place efficiency as our top priority and leave improvements on the efficiency of the designs as future work. Finally, while in this work we provided necessary measures to algorithmically protect masking schemes in typical scalar cores and we showed these measures can lead to secure designs, the properties are not yet sufficient. There are leakage sources which are not covered as-is such as transition leakage that stems from overwriting an old value in memory by its updated version from the register file or glitch related leakages in case of a non-uniform register file or memory due to unknown registers holding a temporary value.

Acknowledgements

We thank Vincent Rijmen, Svetla Nikova, Ventzislav Nikov, and Lennert Wouters for the interesting discussions. This work was supported by CyberSecurity Research Flanders with reference number VR20192203. Siemen Dhooghe is supported by a PhD Fellowship from the Research Foundation – Flanders (FWO).

References

[BBFG18] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. maskverif: a formal tool for analyzing software and hardware masked implementations. *IACR Cryptol. ePrint Arch.*, page 562, 2018.

[BDF+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 Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *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*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[BDP+11] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. Keccak implementation overview, 2011. https://keccak.team/files/Keccak-implementation-3.2.pdf. Retrieved on September 13th, 2022.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[BGG+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[BGI+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 Jesper Buus Nielsen and Vincent Rijmen, editors, *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*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BNN+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small s-boxes. *Cryptogr. Commun.*, 7(1):3–33, 2015.

[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 Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop,*

COSADE 2018, Singapore, April 23-24, 2018, Proceedings, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Dae17] Joan Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2017.

[DDE+20] Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Florian Mendel, and Robert Primas. Protecting against statistical ineffective fault attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):508–543, 2020.

[DNR19] Siemen Dhooghe, Svetla Nikova, and Vincent Rijmen. Threshold implementations in the robust probing model. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 30–37. ACM, 2019.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[FGP+17] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults and the robust probing model. *IACR Cryptol. ePrint Arch.*, page 711, 2017.

[GHP+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 Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Joshua Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance, 2011. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf. Retrieved on March 31th, 2022.

[GMPO20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):152–174, 2020.

[GOP21] Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. *IACR Cryptol. ePrint Arch.*, page 794, 2021.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In Mehdi Tibouchi and Huaxiong Wang, editors, *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*, volume 13091 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2021.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *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*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 199–216. USENIX Association, 2017.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):175–220, 2022.

[Newa] NewAE. Cw308t-stm32f reference. https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F/. Retrieved on April 4th, 2022.

[Newb] NewAE. Newae. cw308 ufo. https://rtfm.newae.com/Targets/CW308%20UFO/. Retrieved on February 25th, 2022.

[NIS15] NIST. Fips pub 202, sha-3 standard: Permutation-based hash and extendable-output functions, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. Retrieved on March 14th, 2022.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

[SBM18] Pascal Sasdrich, René Bock, and Amir Moradi. Threshold implementation in software - case study of PRESENT. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2018.

[Sch] Martin Schläffer. Masked ascon software implementations. https://github.com/ascon/simpleserial-ascon#2-rotated-shares-with-device-specific-fixes. Retrieved on september 27th, 2022.

[SiFa] Inc. SiFive. Sifive fe310-g002 manual v19p04. https://sifive.cdn.prismic.io/sifive%2F9ecbb623-7c7f-4acc-966f-9bb10ecdb62e_fe310-g002.pdf. Retrieved on February 25th, 2022.

[SiFb] Inc. SiFive. Sifive software. https://www.sifive.com/software. Retrieved on January 4th, 2022.

[SM21] Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes nullifying fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):305–342, 2021.

[SSB+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.

[STM] STMicroelectronics. Stm32f415rg reference. https://www.st.com/en/microcontrollers-microprocessors/stm32f415rg.html. Retrieved on April 4th, 2022.

[tea] PULP team et al. Ibex risc-v core. https://github.com/lowRISC/ibex. Retrieved on December 15th, 2021.