IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 4, pp. 133–155. DOI:10.46586/tches.v2024.i4.133-155

Another Evidence to not Employ Customized Masked Hardware

Identifying and Fixing Flaws in SCARV

Felix Uhle[1], Florian Stolz[1] and Amir Moradi[2]

[1] Ruhr-Universität Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
firstname.lastname@rub.de
[2] Technische Universität Darmstadt, Darmstadt, Germany
firstname.lastname@tu-darmstadt.de

Abstract. As a well-studied countermeasure against side-channel analysis attacks, there is a general interest in applying masking to different cryptographic functions executed on different platforms. On the one hand, despite their high performance, masked hardware implementations are dedicated to specific algorithms, making them inflexible. On the other hand, applying masking on software involves serious challenges including significant overhead in terms of efficiency and difficulties to maintain theoretical security guarantees in practice. As a result, a line of research has been devoted to enable masked operations in flexible platforms (i.e., microprocessors) by including some masked modules in their hardware, hence a combination of flexibility and performance. In such scenarios, RISC-V is a natural choice as hardware can be adjusted to the extended instruction set. One such attempt presented at CHES 2021 is known as SCARV, which extends the Instruction Set Architecture (ISA) of a RISC-V core with a rich number of first-order masked operations on both Boolean and arithmetic masked operands. In this work, we conduct a comprehensive analysis of SCARV. Instead of relying on empirical measurements to demonstrate security, we utilize tool-assisted evaluations. Through these evaluations, we identified a couple of design flaws that lead to leakage in the masked implementations hosted by the corresponding processor. These flaws are primarily due to the lack of composability of cascaded components. While heuristic and ad-hoc design principles can result in secure, small, and efficient designs, they lack formal security proofs, which may lead to security flaws, like those we report here. Consequently, this work serves as a motivation for using composable masked modules and tool-assisted evaluations when constructing complex circuits.

Keywords: Masking · Hardware/Software co-design · RISC-V · SCARV

1 Introduction

Numerous Side-Channel Analysis (SCA) attacks against secure cryptographic algorithms have been proposed, exploiting the correlation between device physical characteristics and processed data values. One SCA attack vector is the analysis of the power consumption while processing secret data.

To prevent such attacks, countermeasures have been introduced to dissociate processed data from physical characteristics. An advantage of masking over other countermeasures, such as hiding, is that the security of a masked implementation can be formally proven within a specified attacker model, rather than relying solely on empirical evaluations. Various masking schemes exist, with Boolean masking being notable, especially for symmetric

Received: 2024-04-15 Accepted: 2024-06-15 Published: 2024-09-05

ciphers. Given that symmetric cipher operations are predominantly Boolean, Boolean masking facilitates comparably efficient masked implementations of symmetric ciphers.

All masking schemes necessitate data sharing, leading to a linear increase in the amount of data to be processed. This increase depends on the chosen security order, resulting in varying overheads across different implementations. Two approaches are typically employed to handle this overhead: First, increasing the area of the circuit involved in computation. This allows for parallel processing of the shared data representation, mitigating higher latency. Second, keeping the area constant but processing the shared data sequentially, leading to higher latency (more cycles) due to the increased amount of data to be processed compared to a non-masked design. Both strategies result in higher energy consumption. Whether increasing the area or the number of cycles, more state changes of transistors occur, inevitably leading to increased energy consumption.

Both software and hardware platforms can host masked implementations. Hardware masking, which has extensively grown during the last decade, offers designs with provable security features that maintain the desired security level in practice. However, they lack flexibility and cannot be manipulated after fabrication unless implemented on FPGAs. Software masking, on the other hand, offers flexibility, but suffers from two main drawbacks: 1) very high latency overhead regarding the number of clock cycles [GR17, SBM18, GD23], and 2) inconsistency between theory and practice. In other words, the designs which are proven to be secure in theory often exhibit leakage in practice [BWG+22], originating from the micro-architecture of the processor, which is typically unknown for commercial devices. Solutions to these shortcomings can be categorized into three categories: First, addressing leakage by modifying the assembly code of the implementation and re-evaluating empirically [SCS+21], albeit in a somewhat heuristic manner. Second, considering a conservative model for the micro-architecture to avoid micro-architectural leakages. This, however, leads to extreme overhead [ZM24]. Third, constructing a customized processor to facilitate masked operations, typically by extending the instruction set to include new operations for working on masked operands. These operations are implemented in hardware, providing the flexibility offered by the microprocessor, while the latency can be highly mitigated.

In this work, our focus is on the third category. More precisely, we concentrate on an instruction set extension for RISC-V platforms to support masked operations published at CHES 2021 [GGM+21].

As we elaborate later in Section 2.3, creating masked implementations is a time-consuming, and complex task. In the case study in [MM22], several designs crafted by security experts with years of experience were discovered to still exhibit unexpected leakage. Many flaws go unnoticed due to the limitations of experimental evaluations. Experimental results may vary when employing different measurement setups or making minor design changes, such as altering the placement and routing of masked hardware designs or using different microprocessors (even from the same family but produced by different vendors) in masked software implementations.

Several tools have been developed recently to check the security of circuits under specific security models, such as robust probing or random probing models, aiming to assist designers in evaluating masked designs and addressing the limitations of experimental circuit evaluations [BBD+15, BGI+18, KSM20, BMRT22, RBFSG22]. However, many of these tools are limited to analyzing small implementations and gadgets due to their exhaustive analysis approach, rendering them impractical for more complex designs. One recent tool capable of simulating complex designs, due to its incomplete nature, is PROLEAD [MM22], which was utilized for many experiments in this work.

In the past, the absence of such tools made computer-assisted studies of complex designs impossible. As a consequence, security claims for complex circuits often relied solely on empirical evaluations. This work highlights that tool-assisted evaluations can detect flaws

that empirical evaluations may overlook. This is particularly important for complex circuits designed for performance and efficiency without using composable components. Based on our findings, we recommend employing composable circuits for designing complex systems due to the challenges associated with analyzing such circuits.

Our Contributions. In a tedious process spanning several months, we conducted an in-depth analysis of the masked ALU proposed by [GGM+21]. During this analysis, we revealed a couple of vulnerabilities in the masked ALU, each contributing to leakage in masked operations. We provide insights into why certain constructs used in the design are prone to leaking. Subsequently, we address all identified vulnerabilities and perform an incomplete verification of the security using the glitch and transition extended robust probing model with PROLEAD [MM22]. Furthermore, we demonstrate through examples the critical importance of verifying even the smallest components of a design to ensure correct masking. This underscores the necessity of thoroughly verifying masked circuits, especially when combining modules that have not been proven to be composable, as such combinations can lead to insecure implementations.

2 Background

2.1 Masking

One way to implement a cryptographic algorithm secure against SCA attacks is to apply masking. The concept of masking was introduced in [CJRR99] leveraging Adi Shamir's renowned secret sharing scheme [Sha79]. Following the fundamental principles of secret sharing, the concept of masking involves the fragmentation of secret information into random and independent shares. Thus, only if all shares are known, information about the secret is uncovered. From a cryptographic engineering perspective, masking decouples the processed data from their physical characteristics (on a specific device). One advantage of masking over other countermeasures, e.g., hiding, is that the security of a masked implementation can be proven in a given (attacker) model and not only based on empirical evaluations [MOP07]. For symmetric ciphers Boolean masking is the most common choice to efficiently implement a masked realization of an algorithm. This makes it an important candidate to be implemented as part of an Instruction Set Extension (ISE) with the purpose of accelerating the execution of (symmetric) cryptographic algorithms.

Boolean Masking. Boolean masking was presented as a countermeasure to SCA in [CJRR99]. A Boolean masked sensitive information $x \in (\mathbb{F}_2)^n$ is represented with $k \geq 2$ independent random shares $x_i \in (\mathbb{F}_2)^n$ such that $x = \bigoplus_{i=0}^{k-1} x_i$. Usually, a Boolean sharing of x is generated by sampling $k-1$ shares x_i uniformly at random from $(\mathbb{F}_2)^n$. The remaining share is then given by $x_{k-1} = (\bigoplus_{i=0}^{k-2} x_i) \oplus x$. Thus, knowing $d < k$ shares is not enough to reconstruct x. A Boolean masked algorithm has to compute all operations involving the sensitive information x on its shared representation $\{x_0, \ldots, x_{k-1}\}$.

2.2 *d*-Probing Model

As stated, security of masked implementations can be formally evaluated by means of various adversary models. The d-probing model [ISW03] offers a high level of abstraction, simplifying its usability in comparison to other models [CJRR99, DFS15, PR13]. This ease of use is due to the abstraction of inherent physical characteristics of circuits. Therefore, it is a prominent model to prove SCA resistance.

The model operates under the assumption of *ideal* circuits and defines an adversary capable of measuring up to d freely chosen, noise-free, and *stable* signals. Hence, the model allows the adversary access up to d intermediate values. Security within this model

depends on the adversary's inability to learn information about the sensitive data processed. This, in turn, requires that the distribution of the processed sensitive information remains independent of the distribution across the d intermediate values. The standard d-probing model was expanded to the robust d-probing model [FGMDP+18] to cover the imperfections in hardware circuits, deviating from the *ideal*, and acknowledging physical phenomena such as glitches, transitions, and couplings, which can introduce additional information leakage. In the robust probing model, the original d probes are systematically extended and uniquely tailored to the specific physical effect under consideration. These diverse extensions share a common objective: addressing the worst-case scenario in terms of potential information leakage caused by a specific effect.

By extending the d probes in a manner that comprehensively accounts for all theoretically possible information leakage associated with a given physical effect, the model takes a conservative approach. This conservative assumption is pivotal in modeling arbitrary hardware circuits, as it allows the inclusion of all information that may leak. While this level of conservativeness may seem stringent, it enables the modeling of arbitrary hardware circuits without the need for details about the manufacturing process, or physical characteristics, such as routing and placement. A less conservative approach may not be capable of achieving this without incorporating comprehensive details regarding the specific hardware attributes.

Glitches. Glitches manifest as temporary and unintended signal transitions in the combinational logic between two register stages. This physical effect occurs when signals, with different paths and switching delays, arrive asynchronously at a gate. Thus, the gate computes preliminary outputs before all input signals stabilize. In the context of SCA, glitches can lead to unintended signal recombination.

The transient outputs of a gate depend on complex physical properties of the manufactured circuit, rendering it unfeasible to estimate the concrete information, which is leaked. Due to this limitation, the robust probing model extends the original d probes with allowing them to access all stable intermediate values (register outputs and global inputs), which impact the original probed wires. This extension empowers the adversary to compute all possible leakage functions, including those that arise in an arbitrary hardware realization of a circuit, caused by its physical properties. Consequently, the robust probing model is suitable for testifying security in the presence of glitches.

Transitions. Data transition within memory elements, such as registers, allows an adversary to potentially gain knowledge of the current and the new value stored in a memory element. In response to this, the robust probing model permits the adversary to observe two consecutive clock cycles to model the potential leakage resulting from transitions. As a result, a single probe placed on a register output is extended to two probes within the robust probing models, with one measuring the current and one measuring the new value.

Couplings. Coupling refers to a class of physical effects that cause neighboring wires to influence each other. From the SCA perspective, the influence on adjacent wires may lead to unintentional recombination of signals. Since coupling is a spatial effect, the original probing set is extended with probes, which observe the neighboring wire in a given radius to form a coupling-extended probing set. This however needs information about physical realization of the circuits, i.e., its layout, and is not the focus of our work as it deals with designs at Register-Transfer Level (RTL).

(g, t, c)-Robust d-Probing Model. The notion of (g, t, c)-robust d-probing introduced in [FGMDP+18], indicates the physical effects considered within the robust probing model. Specifically, g and t take binary values (0 or 1) to indicate whether glitches (g) and transitions (t) are considered. For couplings, $c \geq 0$ quantifies how many wires influence each other. In the remainder of this work we use the $(1, 1, 0)$-robust d-probing model

to verify SCA resistance of the underlying circuits. Security in this model is achieved if the criteria of the original d-probing model are fulfilled for a glitch and transition extended probing set. For the remainder of this work, the term "security" exclusively refers to security within the $(1, 1, 0)$-robust d-probing model. It is crucial to emphasize that neither d-probing security nor (g, t, c)-robust d-probing security provides any guarantee of composability. Therefore, independently secure components should be composed with care to maintain the desired security [MMSS19, CS20].

2.3 Hardware Masking

In general, two strategies for building masked hardware implementations can be distinguished. On one side is a handcrafted approach which relies mainly on intuition, heuristics, and empirical evaluation. The counterpart of the handcrafted strategy is the gadget-based masking, which utilizes composable security notions and can deliver proofs for the security of the composed circuit.

Handcrafted Masking. The process of manually implementing a masked circuit requires that the masking is applied to the whole circuit. Combining two such circuits may not lead to a secure design. Every small change to the circuit must be evaluated in the context of the entire design. This makes the process of implementing an SCA-resistant variant of even a small algorithm a challenging, time-consuming, and error-prone task. Additionally, it is not trivial to prove that a handcrafted masked circuit is secure under, e.g., robust d-probing model. Therefore, experimental evaluations are often given for such designs instead of proofs. Due to the strong dependency of experimental evaluations on the measurement setup, such evaluations are only valid in the specific context in which the experiments were conducted. As an example, we can refer to two evaluations of the same design reported in [BBA+22] and [LMMRS23].

Gadget-Based Masking. To design an SCA-resistant circuit, a typical scenario is to divide the chosen algorithm into small components, typically atomic Boolean functions. Each function is then replaced with its masked variant called a gadget. If the gadgets fulfill some well-defined composability properties, it can be guaranteed that the combination of the gadgets leads to a circuit secure in the robust d-probing model. Typically, these gadgets are comparably small, hence it is easy to show that they fulfill the desired composability notions. The downside of this technique is its higher area and/or latency overhead compared to the handcrafted designs.

Domain-Oriented Masking (DOM). DOM is a generic hardware masking technique introduced in [GMK16], which keeps all signals belonging to a share domain independent of the other share domains. Therefore, an implementation following the DOM approach can use $d + 1$-shares to achieve d-th order security in the glitch extended probing model. Since the computation of linear functions over Boolean masking involves only shares of one domain, it is trivial to keep the shares separated. The realization of non-linear functions, e.g., an `AND` operation in a Boolean sharing context, is more challenging, since the component functions involve shares from more than one domain.

The DOM `AND` operation, introduced in [GMK16], extends the concept of [ISW03] to accommodate the robust d-probing model by adding a fresh random and a register stage. In short, two designs for a masked `AND` module have been presented: 1) DOM-*indep*, requiring independently shared inputs, and 2) DOM-*dep*, requiring more randomness and area but without any constraints on the sharing of the inputs. In Figure 1 we show the architecture of a first-order DOM-*dep* `AND` with two fresh masks[1]. We should highlight that – although highly secure and efficient – DOM gates do not fulfill any composability properties.

[1] The other DOM-*dep* `AND` design presented in [GMK16] requires three fresh mask bits.

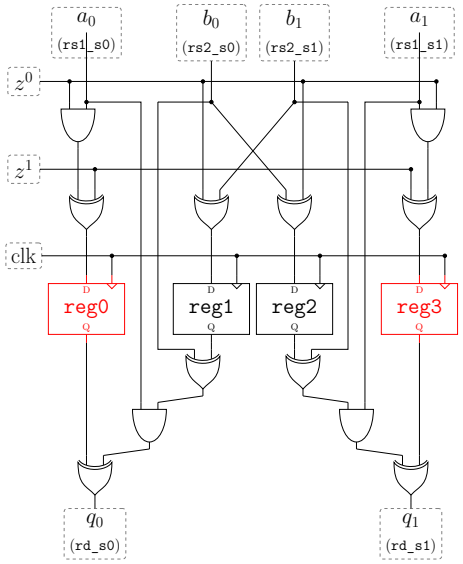


Figure 1: RTL design of an efficient first-order ($d = 1$) DOM-*dep* `AND` gate. z^0 and z^1 denote fresh masks. The labels in parentheses, e.g., rs0_s0, rs1_s1 and rd_s0 follow the notation from [GGM+21] used for inputs to the Arithmetic Logic Unit (ALU). The absence of the red registers in the design and Figure 2a of [GGM+21] results in leakage in the glitch-extended probing model.

2.4 PROLEAD

PROLEAD [MM22] is a simulation-based leakage detection tool designed to analyze the security of masked circuits within the $(1, 1, 0)$-robust d-probing model. To verify the security of a circuit, PROLEAD simulates the circuit with different inputs and examines the dependencies between the secret and the distributions observed by glitch- and transition-extended probes placed on the gate-level netlist of the underlying circuit. Such dependencies are estimated by means of a statistical test, known as G-test [Hoe12].

Although PROLEAD offers the advantage of being able to verify more sophisticated designs compared to formal verification tools such as [BBD+15, BGI+18, KSM20, BMRT22, RBFSG22], its simulation-based approach also introduces a drawback, i.e., incomplete verification. Consequently, the result of the verification by PROLEAD is an approximation, which can be improved by increasing the number of simulations and different input vectors until it attains completeness.

PROLEAD's current version has two limitations: it is incapable of simulating circuits clocked on both negative and positive edges, and it is incapable of simulating dedicated memory modules in system-on-chip and microprocessor designs. The first issue can be addressed by adopting the Device Under Test (DUT) to be clocked on a single edge, which does not affect the characteristics relevant to SCA. The second restriction requires the removal of the memory module. Therefore, leakage caused by the memory cannot yet be evaluated by PROLEAD.

2.5 RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA), aiming for simplicity in both hardware and software through the usage of the Reduced Instruction Set Computer (RISC) principle. In contrast to Complex Instruction Set Computer (CISC) architectures, such as x86, RISC processors implement simple instructions in order to keep the space requirements of the decoder and other functional units relatively small. This in turn

ideally allows instructions to complete in one or two cycles. Furthermore, RISC-V employs modularization to tailor processors for specific use cases. The base ISA, referred to as *RV32I* (for 32-bit architectures), only defines a register set consisting of 32 general-purpose registers (`x0 - x31`) and provides basic integer arithmetic and control-flow instructions. If the designer requires more instructions, additional extensions can be added. For example, the *M* extension includes multiplication as well as division. Linux-capable machines usually implement the *G* extension, which groups many extensions such as the *M* extension together. This modularization, as well as its open-source license, allows researchers to propose and implement own extensions for special purpose acceleration, such as cryptographic computations. The RISC-V standard reserves specific opcode ranges for these custom instructions [PW17].

2.6 Design under Study

In this work, we analyze the instruction set extension for first-order masking proposed by Gao *et al.* in [GGM+21]. The provided instructions allow designers to implement efficient and fast masked implementations in software without utilizing specialized co-processor. They use the SCARV[2] RISC-V core [MP21], which is an advanced embedded-class processor featuring a 5-stage pipeline and implementing RV32I as well as three standard extensions to accelerate bit manipulation, integer multiplication and to allow for compressed instructions. The SCARV project aims to create an open platform which can be used to prototype instruction set extensions that accelerate cryptographic implementations. They also aim for a side-channel hardened processor, which can be achieved by extensions such as the one by Gao *et al.* In their work, the authors extend the core with a random number generator module to generate fresh masks. A Linear Feedback Shift Register (LFSR) with an optional ring oscillator, which provides one true random bit, provides 32 bits of randomness. Furthermore, a *masked ALU* is added, which implements different kinds of *instruction classes*. Each class realizes specific masked operations, such as Boolean masking, arithmetic masking, conversions between the two and acceleration of field operations. The core employs the general-purpose register set to store shares and was modified to make the loading of four values (two operands in shared form) in parallel possible. Accidental share combination is prevented by storing one share of each operand in bit-reversed form. The pipeline was also modified to propagate all operands to the *masked ALU*, where the bit-reversing is eliminated and the requested operation is carried out. Internally, the ALU uses an instantiation of the same LFSR structure described earlier to generate randomness. In their work, Gao *et al.* assess the security of their implementation by performing a Test Vector Leakage Assessment (TVLA).

2.7 Notation

Each submodule of the ALU reduces at most two shared data inputs to one shared data output. The word size of the ALU is given with $l = 32$. a and b are reserved to label the data inputs and q to label the output. $\mathbf{a}$ is a sharing of $a \in (\mathbb{F}_2)^l$ with $\mathbf{a} = (a_0, a_1) \in (\mathbb{F}_2)^l \times (\mathbb{F}_2)^l$ and $a = a_0 \oplus a_1 \in (F_2)^l$. If required, we address the i-th bit $(0 \leq i \leq l)$ of share a_0 with $a_0^{(i)} \in \mathbb{F}_2$. We designate (a_0, b_0) and (a_1, b_1) as the *first* and *second* domain, respectively. The six used masks are given with $z^j \in (\mathbb{F}_2)^l$ with $j \in [0, 5]$. We use parentheses here to distinguish a bit index of a share from an index referring to a specific mask. The i-th bit of a j-th mask is referred with $z^{j,(i)} \in (\mathbb{F}_2)$. Due to the requirements of an iterative circuit, computations of consecutive rounds must be distinguishable. $\dot{a_0}$ is the value of a_0 in the previous round and $\ddot{a_0}$ is the value in the round before that. Gao *et al.* [GGM+21] refer to the global inputs of ALU with `rs1_s0`, `rs1_s1`, `rs2_s0` and `rs2_s1`. The outputs are named `rd_s0` and `rd_s1`. The two shares of each

[2] Available at: https://github.com/scarv/scarv-cpu/tree/scarv/xcrypto/masking-ise

data inputs are distinguished by `_s0` and `_s1`. We follow this scheme to label the inputs and outputs at the boundaries of the ALU.

3 Experimental Analysis I (SCARV)

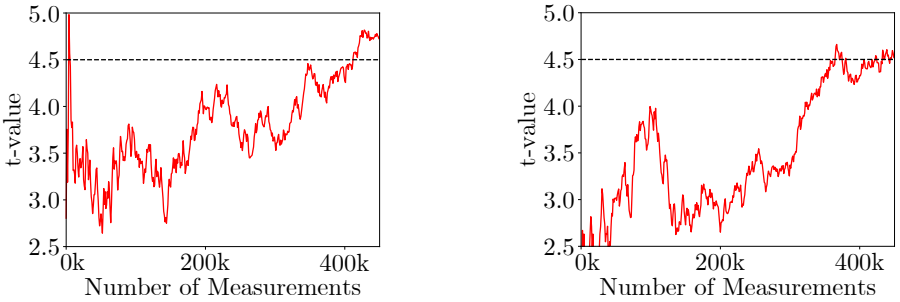
In the following section we detail our measurement setup and show that the original extended SCARV core provided by Gao *et al.* [GGM+21] exhibits leakage by performing a power side-channel analysis.

3.1 Setup

In order to analyze the underlying design, we employed two setups. Both use the latest masked SCARV core as provided by the authors in the `xcrypto/masking-ise` branch of the GitHub repository[2] (commit `b6720e5`). The first setup, referred to as the *SoC setup*, uses the ChipWhisperer CW305 board. This specific board was chosen as it features an Artix `xc7a100tftg256` Field Programmable Gate Array (FPGA), which was also used in the original paper. Furthermore, the authors provide a full System on a Chip (SoC) project for this specific FPGA comprising the Central Processing Unit (CPU) as well as various peripherals, such as General-Purpose Input/Output (GPIO), which we use for setting up a trigger, and a Universial Asynchronous Receiver Transmitter (UART) for sending data. The CPU receives code as well as data from a control computer. The SoC is clocked at 25 MHz by an on-board Phased-Locked Loop (PLL) and its power consumption is measured via a shunt resistor located on the CW305 board. The signal is passed through a 20 dB amplifier and captured by an oscilloscope capable of 2.5 GS/s. Our second setup, referred to as the *low-noise setup*, employs a Sakura-G board, which is specially designed for low-noise measurements. Furthermore, the board is physically located in a room that is shielded from day-to-day noise sources. It features two Spartan-6 FPGA, one acts as a controller and the other one as the device-under-test. The power consumption is captured via a shunt resistor, afterwards the signal is amplified by 21 dB and sampled at 2.5 GS/s. The FPGA is supplied by a 4 MHz clock source. Such a slow clock rate is a common practice in side-channel evaluation as it allows obtaining clear power traces, as the switching noise is considerably reduced. This represents a worst-case scenario for a defender; hence, a secure implementation should not exhibit leakage even under such circumstances. On this setup we only instantiate the *masked ALU* instead of the whole core to eliminate any other potential leakage sources.

3.2 Evaluation

We evaluate the robustness of the original design using a *fixed vs. random* t-test. For the SoC setup, we designed microbenchmarks that only focus on a single instruction of interest as shown in Listing 1. Each benchmark consists of a sequence that activates the trigger, creates a gap via No-Operation (NOP) instructions, executes the desired instruction, and then resets the trigger. A gap between the trigger and the instruction under test is required, as we use a memory-mapped peripheral to set a GPIO pin high. Thus, any consecutive instruction following the store instruction would have already executed and would stay stalled in the pipeline as the load-store unit performs its action. Therefore, we cannot capture the effects created by executing a single masked instruction. Internally a NOP is decoded as `addi x0,x0,0` which is executed just like any other arithmetic instruction, hence creating a short delay, which allows us to activate the oscilloscope in time. Notably, the NOP does not completely clear the micro-architectural state in the pipeline, which is however unproblematic, as each measurement is performed using independently and freshly masked inputs accompanied by fresh masks. To perform the t-test the control computer chooses a fixed or random (masked) value and transmits it to the core. Afterwards, it instructs the core to execute the microbenchmark. For demonstration purposes, we



(a) t-statistic for masked `ADD`. **(b)** t-statistic for masked `AND`.

Figure 2: t-statistic over 450,000 measurements for a masked `ADD` as well as `AND` operation on the SoC setup.

selected the Boolean masked `AND` as well as `ADD` instructions. As shown in Figure 1, the `AND` module misses critical registers, which should lead to detectable leakage. The `ADD` was chosen as a representative of more complex operations, which combine multiple primitives.

```
microbenmark:
li t0, 0x40002000 ; Memory-mapped GPIO used for trigger
li t1, 8 ; Bitmasks to activate/deactivate trigger
li t2, 0
li a5, 0 ; Clear result registers
li a4, 0
nop
sw t1, 0(t0) ; Activate the trigger
nop ; Wait a few cycles for trigger to go high
...
mask.b.add (a5,a4),(a3,a2),(a1,a0) ; Instruction under test
nop ; Wait in order for mask.b.add to pass through all pipeline stages
...
sw t2, 0(t0) ; Deactivate the trigger
ret ; Return
```

Listing 1: Example of a microbenchmark which is used to assess the leakage of the masked addition.

The results are shown in Figure 2. After around 450,000 measurements the t-value crosses the 4.5 boundary with a clear upward trend for both operations, meaning that the fixed and random sets can be differentiated with > 0.99 probability, thus indicating leakage. As our traces are not perfectly aligned and also include noise, we believe that the design leaks with fewer measurements under more ideal conditions. To ensure that the leakage is entirely or in-part caused by the *masked ALU*, we performed the same experiment on the low-noise setup, where the ALU is instantiated without the surrounding core. As we will show later, the randomness generation employed in the original work is not robust. Without changing it, we were able to detect leakage after around 700,000 measurements. The required measurements for both setups differ, because other computational units within the SoC potentially amplify the leakage as explained in Section 5. For comparison reasons, we removed the randomness source and replaced it with a Keccak f-[800] core. Here, the module started to exhibit first-order leakage after 5 million measurements. Figure 3 shows a t-test result after 8 million measurements, where the peaks are more pronounced.

4 Leakage Mitigation

In this section, we analyze the leaking components of [GGM+21], using the latest masked SCARV core as provided by the authors in the `xcrypto/masking-ise` branch of the

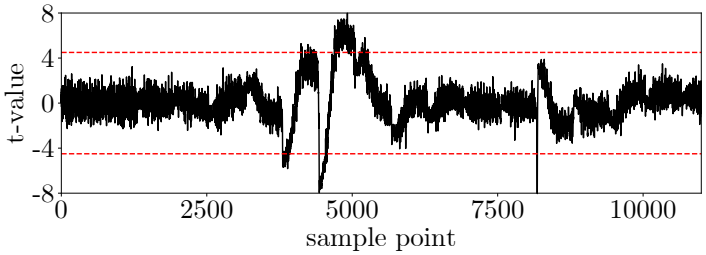


Figure 3: t-test after 8 million measurements for the original masked ALU using an improved randomness engine on the low-noise setup.

GitHub repository[2] (commit b6720e5), and show how to fix them. The extracted modules and the used PROLEAD configurations are available on GitHub[3]. We restrict our analysis to the masked ALU [GGM+21] and within that to the proposed B-class submodule to limit the complexity of this time-consuming process. The modules responsible for field arithmetic (F-class operations) and operations under arithmetic masks (A-class operations) are disabled and removed from the synthesized netlist. Throughout our analysis, we operate under the assumption of ideal masks unless explicitly stated otherwise. We start our investigation by scrutinizing individual components before proceeding to evaluate the combination of these modules.

4.1 `BoolBitwise` Module

The `BoolBitwise` module is used to compute bitwise operations, such as `AND`, `OR`, `NOT`, `XOR`. All operations, except the `OR` operation, are implemented independent of each other. Therefore, within the limits of the `BoolBitwise` module, they cannot interfere and cause leakage. Since bitwise operations process each bit separately, we focus on a single bit and omit positional indices.

The linear operations of the `BoolBitwise` module are implemented in the expected way: The shares of the two domains are never combined. Thus, a first order adversary cannot gain knowledge of the unshared values. Additionally, a non-necessary remasking step is performed as part of the `XOR` operation, but no register is added after the remasking step. Thus, an adversary in the glitch-extended robust probing model learns trivially the used mask z^1. This renders the remasking step ineffective in increasing probing security. In a broader context where the `BoolBitwise` module is incorporated alongside other components, this could potentially even diminish security. This is because z^1 is also utilized in the `AND` operation as illustrated in Figure 1. Taking a conservative approach to avoid potential leakage, we decided to add a register after the remasking step. Another alternative is to completely remove the remasking step, but as we have later verified with PROLEAD, this will lead to leakage when an operation involves the `BoolAdder` module.

The `OR` operation is realized by using the `AND` operation and applying De Morgan's law. However, employing De Morgan's law does not impose any additional security constraints in an SCA setting. This is because it only requires two Boolean negations, which are linear operations and thus do not interfere with the distinct shares. The security of the `OR` follows from the security of the `AND` implementation.

The `AND` operation is implemented in hardware following the DOM-*dep* strategy [GGM+21]. We start our analysis with the implementation of the `AND` operation, as it is utilized multiple times within the masked ALU.

[3] https://github.com/ChairImpSec/scarv-leakage-analysis

First-order optimized DOM-dep AND. Gross *et al.* show in [GMK16, Eq. (12)] a formula enabling the computation of an `AND` operation utilizing only two fresh masks instead of three. [GGM+21, Figure 2a] computes exactly this formula, but lacks two necessary registers to ensure first-order robust probing security. Figure 1 illustrates a non-leaking hardware realization of [GMK16, Eq. (12)]. The red-colored registers are missing in the implementation and visualization of [GGM+21]. The necessity of the red-colored registers to maintain first-order security in the robust probing model is demonstrated in the following.

Probing q_0 (Figure 1) in the glitch-extended robust probing model leads to the probing set

$$P_{q_0} = \{a_0, z^0, z^1, b_0, z^0 \oplus b_1\}$$

with $(z^0 \oplus b_1)$ stored in `reg1`. The elements of P_{q_0} can be used to compute the secret information b with

$$b = z^0 \oplus (z^0 \oplus b_1) \oplus b_0 = b_1 \oplus b_0.$$

Incorporating the red-colored register prevents the adversary from gaining access to the plain mask z^0 through glitches. Consequently, it becomes infeasible to eliminate the mask z^0 from $z^0 \oplus b_1$. As a result, the computation of b becomes unattainable. Having previously developed a generic version of the DOM-*dep* `AND`, we decided to substitute the vulnerable `AND` with our non-leaking generic version, configured with $d = 1$.

4.2 `BoolAdder` Module

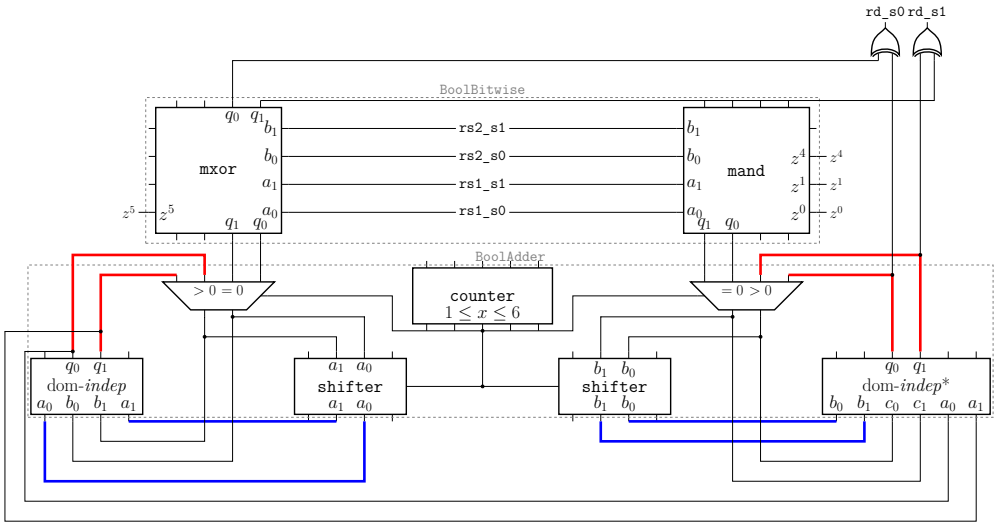


Figure 4: High-level overview of the iterative Kogge-Stone adder (`BoolArith`) implemented in [GGM+21]. Red edges represent signal paths where changes were introduced to address leakage. Blue connections denote alternative positions for introducing countermeasures that were not utilized.

The addition and subtraction operations are implemented using an iterative Kogge-Stone adder [KS73]. The `BoolAdder` module executes the iterative and post-processing steps of the adder, while the necessary preprocessing is handled by the `BoolBitwise` module. Recognition of the interplay between these modules is crucial for conducting an SCA. For this purpose, we introduce a wrapper, which instantiates both modules and connects them in the appropriate way. We refer to this wrapper in the following with `BoolArith` module. The `BoolArith` module allows us to focus on the relevant parts

and exclude effects related to further dependency of other components of the circuit. A high-level overview of the wrapper is given in Figure 4. All wires, which are drawn, represent 32 bit of the same share, e.g. $a_0 = (a_0^{(31)}, a_0^{(30)}, \ldots, a_0^{(0)}) \in \mathbb{F}_2^l$, where a_0 is one of the shares of $\mathbf{a}$. Note that the wrapper requires six random bits, whereas the original design only necessitates four. This additional requirement originates from the inclusion of a generic DOM-*dep* `AND` gate, which demands one bit of randomness per data bit more compared to the initial used optimized DOM-*dep* `AND` version. Another mask (z^5) is added to exclude one source of leakage which we are going to discuss later in Section 4.2.

As illustrated in Figure 4, the iterative part of the computation of a 32-bit addition involves six components.

- Two DOM-*indep* modules: One module strictly adheres to the pipelined design given in Figure 2 [GMK16]. The other module is slightly modified to incorporate an additional `XOR` operation into the DOM-*indep* module. In Figure 4, the modified module is denoted as DOM-*indep**.
- One `counter`: This component controls the multiplexer and the `shifter`. The `counter` is realized as 6-bit shift register, counting the adder cycles from $x = 1$ (000001) to $x = 6$ (100000) using one-hot encoding. The connection between the iterations of the adder, the counter state, and the impact on the `shifter` can be taken from Table 1.
- Two `shifter` components: These are used to interleave the 32-bit iterative intermediates, following the Kogge-Stone approach. The inputs connected to the `shifter` are always shifted by at least one bit position. The number of shifted bits is controlled by the `counter`. Table 1 outlines the connection between the counter state and the number of shifted bits.
- Two multiplexers: Used to distinguish the inputs of the first cycle of the `BoolAdder` from those in subsequent cycles. Each multiplexer selects one of two groups, each containing both shares of a computation result, and forwards the chosen group. In the first cycle ($x = 1$), the preprocessed inputs are fed into the DOM-*indep* logic. Specifically, one input sharing ($\mathbf{b}$) of the DOM-*indep* components is directly driven by the result ($\mathbf{q}$) of the corresponding computation of the `BoolBitwise` operation, while the other sharing ($\mathbf{a}$) is linked to the shifted result ($\mathbf{q} << y$). In the remaining cycles ($x > 1$), the outputs of the DOM-*indep* modules are fed to the inputs of both the `shifter` and the DOM-*indep* modules. This means, one input sharing ($\mathbf{b}$) of the DOM-*indep* modules is directly fed from the output sharing ($\mathbf{q}$) of the same DOM-*indep* component, while the other input sharing ($\mathbf{a}$) is fed by a shifted version of the output sharing ($\mathbf{q} << y$). This is illustrated in Figure 4 and Table 1.

The `mxor` and `mand` module are part of the `BoolBitwise` module, computing the `AND` and `XOR` operation and utilized in the preprocessing step of the `ADD` and `SUB` computation.

First-order flaw. The `BoolAdder` module incorporates a single register stage, integrated within both the DOM-*indep* and DOM-*indep** modules. By way of illustration, we

Table 1: This table illustrates the number of shifted bits, which depends on the state of the `counter`.

Iteration of Adder (x)	Counter State	#Shifted Bits (y)
1	000001	1
2	000010	2
3	000100	4
4, 5, 6	001000, 010000, 100000	8, 8, 8

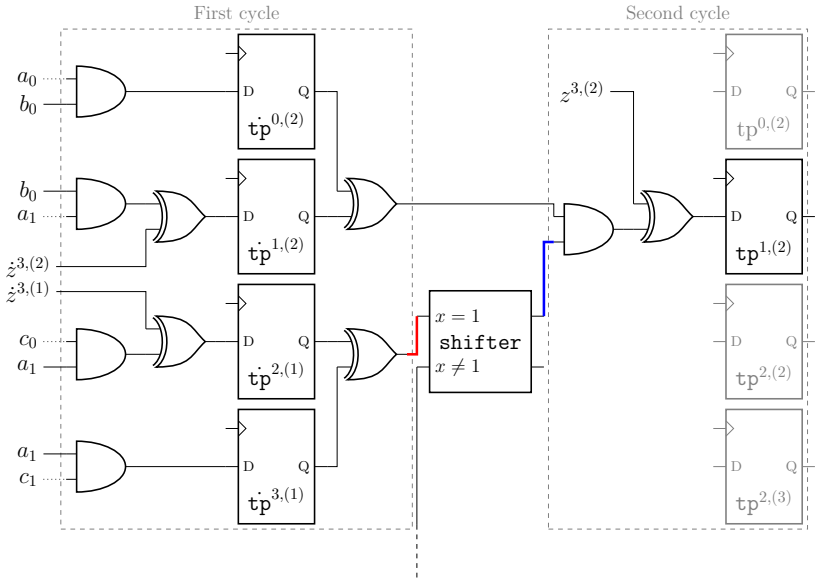


Figure 5: Computation of the second bit of the DOM-*indep* module in the second iteration ($x = 2$) of the `BoolAdder`. The dotted lines are used to emphasize that the signal is propagated through the shift module leading to the visualized connection. The red line represents the signal path where changes were introduced to address leakage. The blue connection denotes an alternative position for introducing countermeasures that were not utilized.

demonstrate through an example that the proposed design fails to meet the criteria for first-order glitch-extended probing security. To provide clarity, our attention is directed towards the computation of the second bit of the $\mathtt{tp}^1$ register within the DOM-*indep* module during the second iteration of the `BoolAdder`. Figure 5 illustrates the components implicated in the signal propagation, which provide a glitch-extended probing set sufficient to observe leakage.

The extension of the probe along the depicted connections is feasible solely due to the fact that the only register stage involved pertains to the DOM-*indep* module, allowing glitches to propagate through the `shifter`. It is worth noting that all visualized components linked to the port of the `shifter` labeled with $x = 1$ do not affect the actual computation. This is intentional, as we are analyzing the second iteration ($x = 2$) of the `BoolAdder`, and assume that the signal forwarded by the `shifter` is shifted by two positions (refer to Table 1). However, the illustrated connections accommodate a signal shifted by only one bit, corresponding to $x = 1$. The notation $x \neq 1$ pertains to the remaining three shifts of the `shifter` module, whose computation is not illustrated, as it is unnecessary for explaining the occurrence of leakage. The inputs connected to the first cycle's gates by dotted lines represent signals that actually propagate to the `shifter` ($x = 1$). However, due to space constraints, the depiction of these `shifter` modules is omitted.

Following the illustrated connections, a probe placed on $\mathtt{tp}^{1,(2)}$ register can be extended to, the probing set

$$P'_{\mathtt{tp}^{1,(2)}} = \{(a_0 \cdot b_0), ((b_0 \cdot a_1) \oplus \dot{z}^{3,(2)}), ((c_0 \cdot a_1) \oplus \dot{z}^{3,(1)}), (a_1 \cdot c_1)\}. \tag{1}$$

We subsequently reduce this set to

$$P''_{\mathtt{tp}^{1,(2)}} = \{P^1_{\mathtt{tp}^{1,(2)}}, P^4_{\mathtt{tp}^{1,(2)}}\} = \{(a_0 \cdot b_0), (a_1 \cdot c_1)\}. \tag{2}$$

The two removed probes reveal no information because $\dot{z}^{3,(2)}$ and $\dot{z}^{3,(1)}$ are fresh and

Table 2: Histogram of the joint distribution of $P^1_{\texttt{tp}^{1,(2)}}$ and $P^4_{\texttt{tp}^{1,(2)}}$ based on the value of a.

(P_0, P_4)	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$a=0$	5	1	1	1
$a=1$	4	2	2	0

Table 3: Histogram of the joint distribution of $F^1_{\texttt{tp}^{1,(2)}}$ and $F^4_{\texttt{tp}^{1,(2)}}$ based on the value of a.

$(F^1_{\texttt{tp}^{1,(2)}}, F^4_{\texttt{tp}^{1,(2)}})$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$a=0$	16	16	16	16
$a=1$	16	16	16	16

independent masks. The histogram for the joint distribution of the remaining probes for a fixed $a \in F_2$ is given in Table 2. The effect of this flaw in a simulation based probing security assessment conduct via PROLEAD is shown in Figure 6a.

Leakage mitigation. The described leakage can be mitigated through two distinct strategies:

1. By preventing the probe extension to all possible shifts of the `shifter` component, achieved by adding a register stage after the `shifter` in the blue marked data path of Figure 4 and Figure 5.

2. By avoiding the probe extension to the four individual component functions of the DOM-*indep* module, accomplished by adding registers after the compression stage in the red marked data path of Figure 4 and Figure 5.

We opt for the second approach to minimize the number of added registers and the changes required to adopt the control logic. Recall, that the leaking probing set, as given in Equation (2), is constructed by observing two of the four individual component functions that contribute to the result of the second iteration of the `BoolAdder`.

Adding a register stage after the compression layer leads to the probing set

$$\begin{aligned} F_{\texttt{tp}^{1,(2)}} &= \left\{ F^1_{\texttt{tp}^{1,(2)}}, F^4_{\texttt{tp}^{1,(2)}} \right\} \\ &= \left\{ \left((a_0 \cdot b_0) \oplus \left((b_0 \cdot a_1) \oplus \dot{z}^{3,(2)} \right) \right), \left(\left((c_0 \cdot a_1) \oplus \dot{z}^{3,(1)} \right) \oplus (a_1 \cdot c_1) \right) \right\}. \end{aligned} \tag{3}$$

The corresponding histogram is depicted in Table 3. Probing the added register enables access to the component functions of $F^1_{\texttt{tp}^{1,(2)}}$ and $F^4_{\texttt{tp}^{1,(2)}}$ respectively. It is evident that both probing sets reveal no information, as one component function of each probing set is perfectly random due to the mask involved in the computation. To prevent transitional leakage, we opt to reset both the added post-compression-layer register and the original DOM-*indep* register (`tp` register in Figure 5) when the captured value is not required in the next clock cycle. Therefore, the two register stages are reset alternating. We adopt the modifications of the DOM-*indep* module to the DOM-*indep** component.

We verified the resulting circuit using PROLEAD in a glitch and transition enabled setting. PROLEAD is configured with default statistical parameters as outlined in [MM22]. Specifically, PROLEAD processes a sufficient number of simulations to detect all effects with an effect size $\phi = 0.01$ and a false-positive probability of $\beta = 10^{-5}$.

The results obtained from PROLEAD are presented in Figure 6b. In the plot, the minimum p-values are depicted in black on a $-\log_{10}(p)$ scale. The red horizontal line represents the false-positive threshold β. The null hypothesis is rejected if the $-\log 10(p)$ value exceeds this threshold, indicating leakage in the $(1,1,0)$-robust 1-probing model.

The gray line represents the quotient of the number of evaluated simulations and the number required to detect all effects with effect size ϕ and false-positive probability β. The blue horizontal line signifies the threshold where the number of evaluations equals the number of required ones. If the gray line exceeds this threshold, PROLEAD has conducted a sufficient number of simulations. Figure 6b shows that our version of the `BoolAdder` in combination with our version of the `BoolBitwise` computes an `ADD` instruction without exhibiting any first-order leakage.

Incompatibility of `BoolBitwise` and `BoolAdder`. As discussed in Section 4.1, the security of the `BoolAdder` relies on the implementation of the `BoolBitwise` module. More precisely, the `mxor` circuit outlined in [GGM+21] lacks a crucial register after the masking step, rendering it unsuitable for non-leaking operation alongside the `BoolAdder`.

This deficiency is exhibited in Figure 6c through the simulation of an instantiation of the `BoolArith` module, comprising our version of the `BoolAdder` and the original proposed version of the `BoolBitwise` module. For this simulation, we set the effect size $\phi = 0.3$, as this is sufficient to detect leakage. The resulting p-value exceeds the threshold by multiple orders of magnitude, indicating leakage.

Additionally, we conduct five supplementary experiments to investigate the impact of reusing a mask and one experiment avoiding the remasking step of the `XOR` operation. In each experiment, we replace z^5 with one mask from the set $Z = \{z^0, z^1, z^2, z^3, z^4, 0\}$ and employ PROLEAD to verify the security of the circuit. These experiments reveal that substituting z^5 with any masks of Z results in a circuit that violates the requirements for a first-order security in the robust probing model.

Flawed LFSR-based random generator. Up to this point, we have assumed uniformly distributed random and independent masks. To ensure this we have utilized PROLEAD to generate the masks in the experiments above. In the following, we investigate the effect of the proposed randomness generator outlined in [GGM+21], employing our non-leaking versions of the `BoolArith` module.

The randomness generator proposed in [GGM+21] is based on 32-bit LFSRs. In each clock cycle, each LFSR is used to generate up to 64 bit randomness. For this purpose, all 32 bits of the current state r and of the next state $\hat{r}$, given by

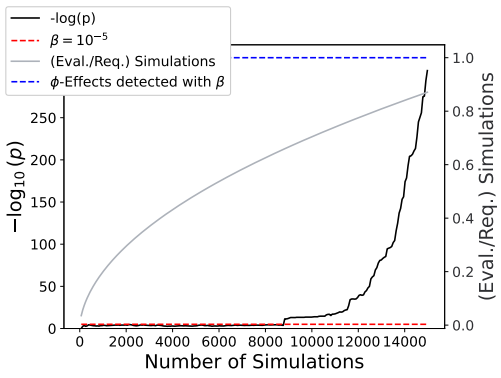
$$r = \begin{cases} c & \text{, first cycle} \\ \dot{\hat{r}} & \text{, otherwise} \end{cases} \tag{4}$$

$$\hat{r} = (r << 1)||(r^{(31)} \odot r^{(21)} \odot r^{(1)} \odot r^{(0)} \oplus tr), \tag{5}$$

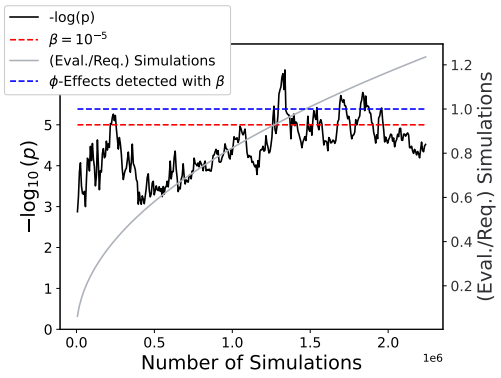
are used in each clock cycle. Here, tr is a true random bit, c is a constant used to initialize the LFSR and $\dot{\hat{r}}$ is the next state $\hat{r}$ of the previous cycle. To assess the impact of the proposed randomness generator, we connect the four random masks z^0, z^1, z^2, z^3 to the two instantiations of the suggested LFSRs. Each LFSR is initialized with different random seeds c generated by PROLEAD. Moreover, the tr bits of both LFSRs are independently generated by PROLEAD.

Once again, the p-value exceeds the threshold by several orders of magnitude, as demonstrated in Figure 6d. This finding indicates that the random number generator utilized by Gao *et al.* is unsuitable for achieving first-order robust probing security.

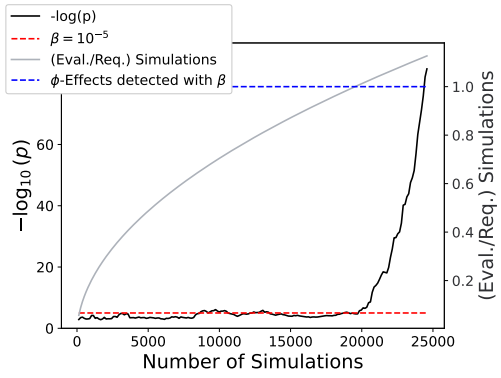
As shown by [CMM+24] it is known that LFSR based randomness generation can lead to insecurity and a better alternative is to use an unrolled implementation of the Trivium Cipher [DC06, DCP08]. For our experiments we continue to use the randomness generated by PROLEAD.



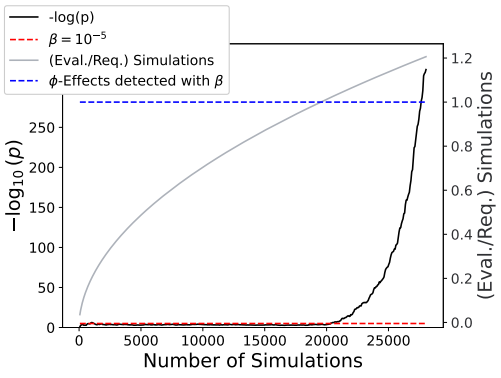
(a) Original `BoolArith` (`ADD`).



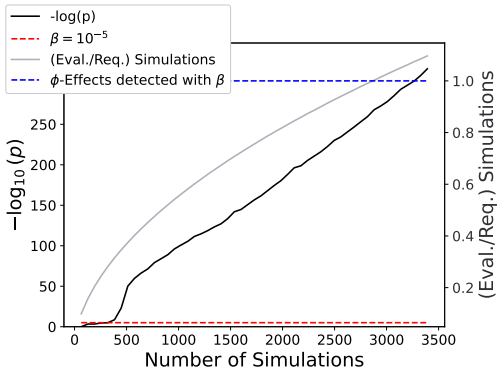
(b) Our modified `BoolArith` (`ADD`).



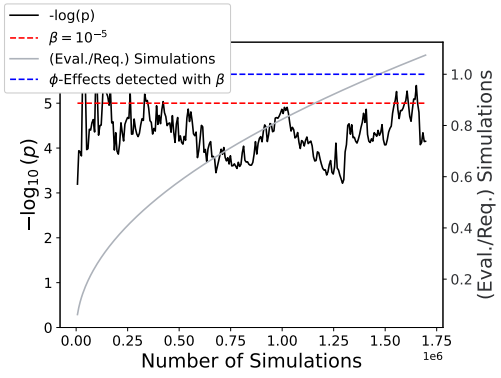
(c) Modified `BoolArith` (`ADD`) with original `XOR` computation.



(d) Modified `BoolArith` (`ADD`) with original randomness generator.



(e) Original `Bool2Arith`.



(f) Modified `Bool2Arith`.

Figure 6: Results of PROLEAD simulations comparing the original design from [GGM+21] (leaking) with our modified design (non-leaking). The red line denotes the false-positive threshold set to $\beta = 10^{-5}$, while the blue line represents the number of traces needed to detect effects with effect size ϕ. When the gray line surpasses the blue threshold, PROLEAD has processed sufficient traces to detect effects with the desired effect size. This is particularly relevant for demonstrating the security of a given design. We infer leakage when the black line consistently exceeds the red line.

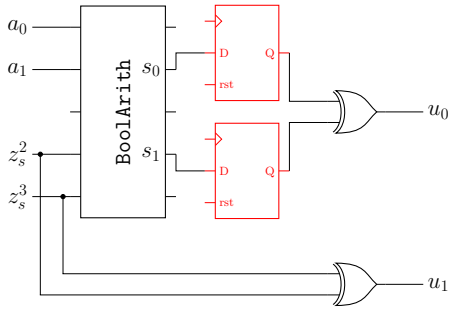


Figure 7: Circuit involved in the computation of the `Bool2Arith` instruction. The red registers, although not originally part of the circuit, are necessary to prevent leakage. s_0 and s_1 denote the outputs of the `BoolArith` module and u_0 and u_1 are both shares of the computed arithmetic sharing $\mathbf{u}$. z_s^2 and z_s^3 remain fixed throughout the computation of the arithmetic sharing.

4.3 `Bool2Arith` Operation

The `Bool2Arith` operation[4] converts Boolean masked values to arithmetic masked values. Although not part of the B-class operations, we analyze it as it solely involves operations from the B-class submodule. This operation takes both shares (a_0, a_1) of a Boolean sharing $\mathbf{a}$ and computes the arithmetic sharing $\mathbf{u} = (u_0, u_1) = ((a_0 \oplus a_1) + (z^2 \oplus z^3), (z^2 \oplus z^3))$. The corresponding circuit is shown in Figure 7.

The red-colored registers are not part of the original `Bool2Arith` operation. We simulate the circuit, without these red registers using PROLEAD with $\phi = 0.5$. In the simulated circuit we utilize our non-leaking version of the `BoolArith` module. However, as demonstrated in Figure 6e, the `Bool2Arith` operation leaks even when computed using our modified version of the `BoolArith` module. This leakage occurs because the adversary gains access to temporary values of both output shares s_1 and s_2 of the `BoolAdder` module. To address this, we introduce the red registers, which are reset until the `Bool2Arith` computation is completed. This prevents the recombination of the input shares by temporary computations. Figure 6f demonstrates by visualizing PROLEAD's simulation results that our adopted version mitigates the leakage. The downside of this approach is that we need one cycle more to compute the instruction.

4.4 Masked ALU

In this section, we investigate the $(1, 1, 0)$-robust first-order probing security of the entire ALU, incorporating the modifications made to the `BoolBitwise`, `BoolAdder`, `BoolArith` and `Bool2Arith` components. Additionally, the ALU comprises two further modules:

1. The `BoolMask` module, responsible for generating a sharing $\mathbf{a}$ of an unshared data input a and performing remasking operations.

2. The `BoolShift` component, capable of computing left and right shifts and rotating a given sharing $\mathbf{a}$ by a specified number of bits.

All these modules are connected to a multiplexer, which selects the expected output based on the chosen `opcode`. The `BoolBitwise` unit computes all bitwise operations when an opcode representing such an operation is active. Due to the multiplexer, an adversary can observe one output share of all bitwise operations. Furthermore, the adversary has access to the inputs of the `BoolMask` module, since it contains no register. The mask used

[4] We should note that the `Bool2Arith` module is different from the `BoolArith` module explained and discussed in Section 4.2.

in the `BoolMask` operation is z^0. Recall that z^0 is also used in the computation of the `AND` operation. Consequently, probing the `rd_s2` output of the ALU while any bitwise operation is computed reveals the partially glitch- and transition-extended probing set

$$P_{\texttt{rd\_s2}} = \left\{ a_1 \cdot \dot{z}^1, \left((a_1 \cdot \dot{z}^0) \oplus \dot{z}^4 \right), b_0 \oplus \dot{z}^0, b_1 \oplus \dot{z}^1, a_1, \dot{z}^0 \right\}. \tag{6}$$

Note, that $\dot{z}^0$ is only accessible through transitions. The vulnerability introduced by $\dot{z}^0$ is shown in the histogram Table 4 of $P_{\texttt{rd\_s2}}$. This issue can be resolved by avoiding z^0 as the mask of the `BoolMask` module.

Table 4: Histogram of the joint distribution of $P_{\texttt{rd\_s2}}$ based on the value of b.

$(P_{\texttt{rd\_s2}})$	0	1	2	3	4	5	6	7	8	9
$b = 0$	6	22	22	12	12	12	4	8	10	4
$b = 1$	8	20	22	12	10	14	10	2	4	10

We opt to replace z^0 with z^5, as the later one is solely utilized in the remasking step of the `XOR` operation. While this adjustment suffices to secure the computation of a single `AND` operation within the ALU circuit, it falls short when different operations with distinct opcodes, but identical data inputs are executed consecutively. Rather than tackling each individual issue arising from the various combinations of subsequently executed operations, we choose to address them collectively.

Recall that while we have individually demonstrated the security of each module, we cannot directly infer the security of the entire ALU from the security of its constituent components. This limitation arises because the output shares of different modules can potentially be probed simultaneously through a single glitch or transition extended probe and no composability notions are used. To circumvent the probe extension to multiple modules, we introduce a new register stage before the inputs of the output multiplexer of the ALU. Each output of the different modules is connected to a register, which is set to zero as long as the opcode and the ready signal of the corresponding instruction are not active. Consequently, the multiplexer is always driven by at most one final output signal, with the other inputs being zero. This prevents the probe extension to different inputs of the multiplexer by glitches, but introduces one cycle overhead for each instruction.

Moreover, since each computation involves at least one register, there is always one clock cycle between two completed operations where all inputs to the multiplexer are zero. This prevents the extension of a probe to different output signals by transitions. Collectively, these measures allow us to conclude the security of the entire ALU from the security of its independent modules. The simulation results for the entire ALU computing an `ADD` are presented in Figure 8.

5 Experimental Analysis II (Fixed ALU)

We now evaluate our improved ALU by performing a power side-channel analysis. We employ the same setup described in Section 3.1 and measure the ALU both in isolation and as an integrated part in the core. As explained in Section 4.2, the randomness generation originally implemented in the SCARV core is flawed. For our measurements, we thus replaced the LFSRs with a more robust Keccak-f[800] core, which generates 576 pseudo-random bits per cycle. First, we instantiated our ALU on the low-noise setup and verified that it is leakage-free by asserting different opcodes and observing the power consumption.

Figure 9 shows the t-value of a masked addition over all sample points after 100 million traces. This operation was again chosen, as the masked addition combines multiple

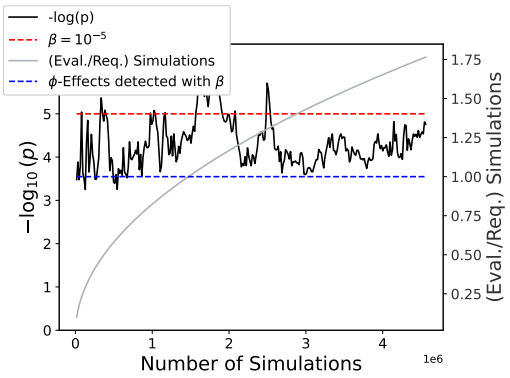


Figure 8: Result of simulating our non-leaking masked ALU (`ADD`) conducted by PROLEAD.

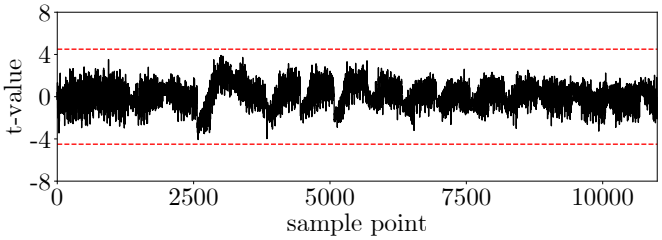


Figure 9: t-test after 100 million measurements for our improved masked ALU on the low-noise setup.

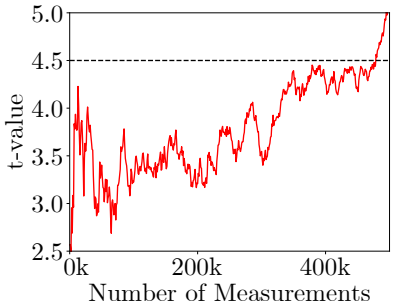


Figure 10: t-statistic over 500,000 measurements for a masked ADD using our improved ALU on the SoC setup.

primitives. Notably, all t-values stay below the threshold of 4.5, indicating no leakage. This is in stark contrast to our results in Section 3.2, where the ALU leaked after only 5 million measurements when performing the same operation. Integration into the SoC setup is straight-forward. We removed the old ALU and replaced it with our improved version, including the Keccak core. No further changes were required, even though our design sometimes requires more cycles, as the pipeline stalls automatically as long as the internal *valid* signal of the masked ALU stays unasserted. The results of running a masked addition microbenchmark can be seen in Figure 10.

Surprisingly, the core still exhibits leakage after around 500,000 measurements. However, as we have above, our ALU does not leak. Thus, there are multiple possible explanations: First, the leakage may be caused by memory operations. Neither PROLEAD nor our low-noise setup took memory operations into account. However, no memory interactions

Table 5: Hardware overhead of the original masked ALU and our improved ALU using a 45nm manufacturing node.

Module	Original ALU	Improved ALU
`Bool2Arith`	208.27 GE	208.27 GE
`BoolAdder`	3235.90 GE	3845.82 GE
`BoolBitwise`	1077.00 GE	2773.31 GE
`BoolShift`	451.13 GE	1285.04 GE
Other	1184.06 GE	3068.80 GE
Total Area	6607.48 GE	11180.24 GE

(e.g., to load shares from memory) take place during or right before the measurement starts. It is therefore safe to assume that the memory is not the main cause of the leakage. Second, our simulations do not consider FPGA specific effects, such as *coupling*, where shares which are routed through physically adjacent wires cause interference and accidental share combination. While this may be the case, we believe these effects to be relatively minuscule and therefore unable to create the significant leakage we observe after only 500,000 measurements. During further inspection of the execution stage of the core we noticed that the operands are propagated to *all* functional units within the pipeline stage. This includes the, for example, normal ALU, but also units responsible for multiplication and resolving conditional jumps. Additionally, the pipeline stage performs a two-stage decoding process. A global select signal decides, which result the pipeline stage should select, i.e., the output of which functional unit should be forwarded. An operation-select signal decides, what operation a functional unit should carry out. The operation-select signal may overlap with other functional units, meaning that the same signal instructs unit X to perform some operation and unit Y to perform some other operation *at the same time*. Therefore, during the execution of a masked operation, all other units will calculate some result using the input shares. The multiplier in particular receives three inputs, of which two are a shared operand. Thus, it has access to both shares of one input value. Even though the masked ALU is the only module with access to the unprotected shares (e.g., eliminating the bit-reversed form), it seems the protected shares still leak information. One obvious edge case may be a shared value of 0, as both shares will combine to 0 regardless of the bit orientation. However, further investigation is required to identify the exact leakage source within the SCARV core, which is out-of-scope for this work.

We furthermore evaluated our solution regarding area as well as timing overhead. For our comparison we employed Synopsys Design Compiler as well Silvaco's Open-Cell 45nm FreePDK. We synthesized the original ALU without the modules, which we did not consider in this work (namely: field multiplication as well as arithmetic masking), and our improved leakage-free version. The results can be seen in Table 5. Notably, the improvement in security comes at the cost of additional logic gates and clock cycles. The most overhead is created by additional register stages within each module. Concerning the timing overhead, it should be noted that the original ALU requires only one clock cycle for most operations except for the Boolean addition/subtraction and `Bool2Arith` operation which takes 6 clock cycles. Our ALU requires two clock cycles for Boolean logic operations, 13 cycles for an addition/subtraction and 14 for a `Bool2Arith` conversion. These overheads are again created by additional register stages, which are required for a leakage-free design.

6 Conclusions

In this work, we conducted a thorough analysis of the masked ALU proposed in [GGM+21], identifying and addressing eight flaws leading to detectable leakage in both simulation

and practice. Two critical flaws, including the weak randomness generator and the incorrectly implemented DOM-*dep* module, could have been easily avoided by using formal verification tools. The remaining six flaws originate from combining independently non-leaking modules, resulting in leakage due to interference. Detecting these flaws with complete formal verification tools is not possible due to computational requirements, necessitating the use of incomplete tools like PROLEAD to check module combinations. However, even with tool support, identifying all flaws is a complex and time-consuming task, and there is no guarantee of detecting all vulnerabilities. This highlights once again the error-prone nature of implementing large circuits solely through customized handcrafted masking approaches. Therefore, employing provably composable gadgets is a preferable approach to implement larger circuits, such as an ALU, and ensure security in the first-order robust probing model. Even the area efficiency of a handcrafted masking strategy for such a complex circuit is questionable, due to the large amount of overhead introduced to mitigate the flaws in the described simulate-fix-repeat procedure. It is worth highlighting that we do not assume that only composable gadgets can lead to secure circuits. In particular, for smaller circuits the handcrafted approach might be preferable due to its potential to produce more compact, efficient, and secure designs.

Furthermore, employing a non-leaking version of the ALU throughout the entire CPU core does not guarantee a non-leaking CPU core. This demonstrates the non-trivial nature of combining masked and non-masked parts of a core. The countermeasures presented in [GGM+21] seem insufficient to prevent leakage. Therefore, we assume that the registers used for masked instructions must be strictly separated from those involved in non-masked instructions. This aspect could be investigated further in a natural follow-up study. One approach could involve replacing the ALU with one constructed based on composable gadgets. With such an ALU, it is possible to delve deeper into the approach of [GGM+21] for combining the masked ALU with the unmasked parts of the CPU, analyzing it in more detail.

Moreover, the discrepancies between the measurements conducted in [GGM+21] and ours underscore the limitations of empirical validation through measurements in verifying the general security of a design, as these measurements are influenced by additional factors.

Acknowledgments

The work described in this paper has been supported in part by the Federal Ministry of Education and Research of Germany through the Project KOSEF (16KIS1597).

References

[BBA+22] Yaacov Belenky, Vadim Bugaenko, Leonid Azriel, Hennadii Chernyshchyk, Ira Dushar, Oleg Karavaev, Oleh Maksimenko, Yulia Ruda, Valery Teper, and Yury Kreimer. Redundancy aes masking basis for attack mitigation (rambam). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):69–91, Feb. 2022.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. *Verified Proofs of Higher-Order Masking*, pages 457–485. Springer Berlin Heidelberg, 4 2015.

[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*, pages 321–353. Springer International Publishing, 3 2018.

[BMRT22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. Ironmask: Versatile verification of masking security. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 5 2022.

[BWG+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. *Provable Secure Software Masking in the Real-World*, pages 215–235. Springer International Publishing, 3 2022.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. *Towards Sound Approaches to Counteract Power-Analysis Attacks*, pages 398–412. Springer Berlin Heidelberg, 12 1999.

[CMM+24] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking – unrolled trivium to the rescue. *IACR Communications in Cryptology*, 1(2), 7 2024.

[CS20] Gaetan Cassiers and Francois-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[DC06] Christophe De Cannière. *Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles*, pages 171–186. Springer Berlin Heidelberg, 2006.

[DCP08] Christophe De Cannière and Bart Preneel. *Trivium*, pages 244–266. Springer Berlin Heidelberg, 2008.

[DFS15] Stefan Dziembowski, Sebastian Faust, and Maciej Skorski. *Noisy Leakage Revisited*, pages 159–188. Springer Berlin Heidelberg, 4 2015.

[FGMDP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89—120, 8 2018.

[GD23] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):155–179, Mar. 2023.

[GGM+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 283–325, 8 2021.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Paper 2016/486, 2016. https://eprint.iacr.org/2016/486.

[GR17] Dahmun Goudarzi and Matthieu Rivain. *How Fast Can Higher-Order Masking Be in Software?*, pages 567–597. Springer International Publishing, 4 2017.

[Hoe12] Jesse Hoey. The two-way likelihood ratio (g) test and comparison to two-way chi squared test. Jun 2012.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. *Private Circuits: Securing Hardware against Probing Attacks*, pages 463–481. Springer Berlin Heidelberg, 2003.

[KS73] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22:786–793, 8 1973.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. *SILVER – Statistical Independence and Leakage Verification*, pages 787–816. Springer International Publishing, 12 2020.

[LMMRS23] Daniel Lammers, Amir Moradi, Nicolai Müller, and Aein Rezaei Shahmirzadi. A Thorough Evaluation of RAMBAM. In *CCS '23: ACM SIGSAC Conference on Computer and Communications Security*. ACM, 11 2023.

[MM22] Nicolai Müller and Amir Moradi. Prolead. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 311–348, 8 2022.

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited: or why proofs in the robust probing model are needed. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):256–292, Feb. 2019.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MP21] Ben Marshall and Daniel Page. Scarv: a side-channel hardened risc-v platform. 2021.

[PR13] Emmanuel Prouff and Matthieu Rivain. *Masking against Side-Channel Attacks: A Formal Security Proof*, pages 142–159. Springer Berlin Heidelberg, 2013.

[PW17] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 1st edition, 2017.

[RBFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. Verica - verification of combined attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 255–284, 8 2022.

[SBM18] Pascal Sasdrich, René Bock, and Amir Moradi. *Threshold Implementation in Software*, pages 227–244. Springer International Publishing, 4 2018.

[SCS+21] Madura A. Shelton, Łukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 11 2021.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 11 1979.

[ZM24] Jannik Zeitschner and Amir Moradi. PoMMES: Prevention of Microarchitectural Leakages in Masked Embedded Software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3), 2024. https://eprint.iacr.org/2024/574.