

# PROLEAD\_SW

## Probing-Based Software Leakage Detection for ARM Binaries

Jannik Zeitschner\*<sup></sup>, Nicolai Müller\*<sup></sup> and Amir Moradi<sup></sup>

Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany

[firstname.lastname@rub.de](mailto:firstname.lastname@rub.de)

**Abstract.** A decisive contribution to the all-embracing protection of cryptographic software, especially on embedded devices, is the protection against Side-Channel Analysis (SCA) attacks. Masking countermeasures can usually be integrated into the software during the design phase. In theory, this should provide reliable protection against such physical attacks. However, the correct application of masking is a non-trivial task that often causes even experts to make mistakes. In addition to human-caused errors, micro-architectural Central Processing Unit (CPU) effects can lead even a seemingly theoretically correct implementation to fail to satisfy the desired level of security in practice. This originates from different components of the underlying CPU which complicates the tracing of leakage back to a particular source and hence avoids making general and device-independent statements about its security.

PROLEAD has recently been presented at CHES 2022 and has originally been developed as a simulation-based tool to evaluate masked hardware designs. In this work, we adapt PROLEAD for the evaluation of masked software, and enable the transfer of the already known benefits of PROLEAD into the software world. These include (1) evaluation of larger designs compared to the state of the art, e.g. a full Advanced Encryption Standard (AES) masked implementation, and (2) formal verification under our new generic leakage model for CPUs. Concretely, we formalize leakages, observed across different CPU architectures, into a generic abstraction model that includes all these leakages and is therefore independent of a specific CPU design. Our resulting tool PROLEAD\_SW allows to provide a formal statement on the security based on the derived generic model. As a concrete result, using PROLEAD\_SW we evaluated the security of several publicly available masked software implementations in our new generic leakage model and reveal multiple vulnerabilities.

**Keywords:** Side-Channel Analysis · Leakage Detection · Software · Masking

## 1 Introduction

Nowadays, researchers greatly understand how to design cryptographic algorithms that guarantee security in a black-box model. However, even two decades after the first introduction of Side-Channel Analysis (SCA) attacks by Kocher et al. [Koc96, KJJ99], the secure implementation of cryptography, assuming a grey-box scenario, is still a complicated task. In particular, the last twenty years have shown a wide range of successful SCA attacks exploiting the physical leakage of cryptographic implementations. In particular, by observing the physical characteristics of the target device, often power consumption [KJJ99] or electromagnetic radiation [GMO01], an adversary can recover secrets stored in and processed by the underlying device. To prevent SCA attacks at the algorithmic level, the *masking* countermeasure, based on secret sharing [Sha79], is a popular approach [CJRR99].

---

\*These authors contributed equally to this work.

If a masking scheme satisfies some basic assumptions, i.e., uniform sharing and sufficient noise, its security becomes abstractable under formal adversary models [ISW03, DDF14].

However, the naive integration of masking only rarely leads to the expected result. Multiple factors at different abstraction levels can cause even a theoretically correct masked implementation to leak. For example, if the masked implementation is provided in a high-level language, the compiler settings, later used, are crucial [BWG<sup>+</sup>22]. Optimizations during compilation can modify the implementation so that basic principles of masking are broken, e.g., share independence. Moreover, cryptographic software executed on an arbitrary Central Processing Unit (CPU) may leak information due to physical defaults occurring on the underlying CPU’s (micro-)architecture [FGP<sup>+</sup>18]. Typical examples include (1) transitions while updating particular registers or the Random Access Memory (RAM) leaking information about the new and the overwritten memory state [PV17], and (2) glitch-related leakage due to the internal hardware architecture of the pipeline or Arithmetic-Logic Unit (ALU) [CGD18]. Micro-architectural leakages are often strongly dependent on the specific CPU being used, which presents a challenge in making security claims that can apply across different CPUs. While it is important to demonstrate the security of software on a particular CPU, this alone is not enough as the software may ultimately be executed on a variety of different CPUs.

In addition to the correct implementation of a masking scheme, it is, therefore, crucial to examine and prove its proper security. In recent years there have been some attempts to evaluate masked software implementations reliably. A detailed summary of existing works is given in Table 1 of [BBYS22]. Usually, their underlying approaches can be broadly divided into two categories with individual drawbacks. Formal verification makes it possible to prove the security of short building blocks such as gadgets or small S-boxes encompassing a low number of instructions. However, proving the security of an implementation consisting of thousands of instructions, e.g., a complete Advanced Encryption Standard (AES) implementation in software, becomes infeasible due to the higher run-time and memory requirements.

Another branch of research focuses on the simulation of leakages. As the simulation considers only a fixed set of input vectors, the evaluation is accelerated significantly but at the cost of lower accuracy. Hence, leakage simulators can evaluate implementations that are out of the scope of formal verification but can only give a preliminary intuition of the leakage instead of a security proof. Moreover, the evaluation applies simple heuristics such as Hamming Weight (HW) or Hamming Distance (HD) and is not in line with formal security models.

At CHES 2022, a combination of formal verification and leakage simulation was presented for the first time. The resulting framework, called PROLEAD [MM22], verifies the security under a formal adversary model by simulating intermediate signals of the target hardware circuit. The simulation-based approach makes it possible to formally analyze even larger designs, e.g., a completely masked AES core under the generic robust  $d$ -probing model. PROLEAD has thus made it possible to detect so far unnoticed vulnerabilities in previously published designs (see Table 2 of [MM22]). However, PROLEAD can only evaluate masked hardware designs and does not cover software implementations.

## 1.1 Contributions

We argue, rather than having only a security statement for a software implementation on a specific CPU, software developers would prefer a broader security statement that is independent of a concrete CPU. Therefore, we design a generic model that includes a variety of leakages and provides security statements in this generic model. We adapt the working principle of PROLEAD to evaluate masked software implementations independent of an actual CPU design. We investigate observed leakages on multiple CPU architectures in public literature. We formalize the found leakage into generic security statements. Those

security statements build the foundation for our new CPU-independent leakage model. Based on this model we define probes that are available to an  $d$ -probing adversary. The resulting framework `PROLEAD_SW` is publicly available via GitHub<sup>1</sup> and extends the original `PROLEAD` with the ability to evaluate the  $d$ -probing security under our generic leakage model for CPUs of any binary compiled for the ARMv6-M, ARMv7-M, and ARMv7E-M architecture. More precisely, we put no restrictions on the code structure, i.e., we are able to handle recursions, loops, conditional branches, and even non-constant time code.

## 1.2 Existing Tools

In the following, we compare `PROLEAD_SW` to the existing works from the open literature.

### 1.2.1 Formal Verification

The purpose of `maskVerif` [BBC<sup>+</sup>19] is to verify higher-order masked software for probing-security, NI [BBD<sup>+</sup>15], and SNI [BBD<sup>+</sup>16] notions under the  $d$ -probing model [ISW03] with transitions. Given a program written in an intermediate representation, `maskVerif` checks based on probabilistic information flow tracking whether leakages are independent of secrets for each possible observation set. The tool is sound and complete for linear equations and reduces the occurrence of false negatives in non-linear cases.

The compiler framework `TORNADO` [BDM<sup>+</sup>20] aims to generate provable secure masked and bit sliced C code for arbitrary orders under the  $d$ -probing model, respectively register probing model [BDM<sup>+</sup>20]. It receives a cipher written in the domain-specific language `Usuba` and verifies the security under the desired attacker model with `TightPROVE+`, an extension of the original `TightPROVE` [BGR18]. If an attack is detected, `TORNADO` introduces refresh gadgets based on sound heuristics.

`COCO` [GHP<sup>+</sup>21] is a white-box verification tool, which maps the verification of masked software to a hardware verification problem. Its underlying formal verification approach is based on `REBECCA` [BGI<sup>+</sup>18], which uses correlation sets and SAT-solver to detect leakage. By providing a gate-level netlist and an assembly implementation `COCO` verifies the secure execution under the time-constrained probing model. This approach enables `COCO` to make formal statements about the implementation's security. Nevertheless, `COCO` cannot evaluate a software with a non-constant control flow.

### 1.2.2 Leakage Simulation

The goal of `ASCOLD` [PV17] is to detect violations of the Independent Leakage Assumption (ILA) by simulated information flow tracking. The tool considers the micro-architectural effects observed on an AVR micro-controller. It operates on masked AVR assembly code with annotated inputs (*random* or *masked*) and propagates them through the entire execution. Hence, `ASCOLD` can guarantee the detection of an ILA violation even if they are caused by memory overwrites (transitions) or neighbor leakage effects. However, not all share re-combinations leak, which makes `ASCOLD` over-conservative.

The intention of `MAPS` [CGD18], a power-simulator for the Cortex-M3 processor, is to examine the security of software designs by considering target-specific leakages, with a particular focus on leakage introduced by hidden pipeline registers. Based on the CPU netlist, which the authors possess, they can model this behavior during the simulation. `MAPS` cross-compiles C code with optional inline assembly to an ARMv7-M binary whose HD leakage is simulated and evaluated by applying Welch's t-test.

`ELMO` [MOW17] simulates the power consumption of Cortex-M0 and M4 processors at the instruction level. It makes use of a power model achieved by performing linear regression on physical measurements. Thus, the goal of `ELMO` is to generate accurately

<sup>1</sup><https://github.com/ChairImpSec/PROLEAD>

simulated power traces tailored to a specific CPU. After generating the power model, ELMO receives a compiled binary as input and simulates the power consumption according to its adapted model. The generated traces can then be evaluated with known leakage assessment techniques. A recent extension ELMO\* [SSB<sup>+</sup>21] further models leakage between non-consecutive instructions, i.e. leakages due to short pipelines, and identifies leaking storage elements.

**RootCanal** follows a white-box approach to identify the origin of SCA leakages, which either come from the executed instructions or the underlying hardware. Given the assembly-level source code and a System-on-a-Chip (SoC) design in a Hardware Description Language (HDL), **RootCanal** performs non-specific architecture correlation analysis on simulated power traces. If leakage is detected, **RootCanal** backtraces the leakage to the actual high-level source, i.e., the executed software instruction or modules in the HDL source.

### 1.2.3 Comparison

In the following, we elaborate on the most important differences between PROLEAD\_SW and the works mentioned above.

**Effectiveness.** Currently, the verification of software can be broadly divided into two approaches. The first approach comes with a great level of precision. In order to provide this precision, tools in this category are specific to a particular CPU or require a specific leakage model. This includes frameworks such as **RootCanal**, **COCO** or **ELMO**. This makes either the adaption of the leakage model to other CPUs non-trivial, e.g., in the case of **ELMO** or requires the possession of hardware netlists, which limits their application mostly to pre-tapeout verification. On the other hand, we are able to choose tools that are more generic, like **TightProve+** or **maskVerif**. They do not require specific netlists or models, but perform their evaluation on higher levels of abstraction. This allows them to perform the verification of software for arbitrary CPUs. The drawback of those tools is the lack of advanced micro-architectural effects or even micro-architectural effects in general. This could potentially reduce the significance of the results in practice. With **PROLEAD\_SW** we position us in between those two strategies. We do not require a specific netlist or dedicated leakage model. We rather introduce a generic leakage model that intends to cover a large set of CPU implementations. While being independent of a specific hardware we introduce a variety of micro-architectural effects to reduce the gap between our generic leakage evaluation and the real world behaviour.

**Performance.** As formal verification is exhaustive, meaning that it checks all possible input vectors, evaluating larger implementations becomes infeasible. As an example, we consider the software case studies of **maskVerif** available online<sup>2</sup>. Most experiments target small S-boxes at low security orders while evaluating the probing security of larger designs, e.g. a complete AES in software is not possible. The same restrictions hold for **COCO**. In particular, the largest implementation in the case studies of **COCO** is a single masked AES S-box running for 1900 cycles. Simulation-based approaches, such as **PROLEAD\_SW**, evaluate even larger designs with acceptable resources, e.g. complete AES masked implementations executing more than 100 000 instructions.

**Confidence.** Formal verification tools, i.e. **maskVerif**, **TightPROVE**, and **COCO**, are free of false negatives. Hence, a positive result proves the security of the implementation under a certain model. However, some formal verification tools such as **maskVerif** and **TightPROVE+** (accepting only abstracted **Usuba** code) evaluate software abstractions. Therefore, both

<sup>2</sup><https://cryptoexperts.com/maskverif/>

tools do not verify the actual instructions executed on a CPU but an abstract representation. As PROLEAD\_SW operates on the binary and its actual instructions, our evaluation is closer to the real world than tools which operate on an abstract representation that does not consider the concrete machine code instructions or operate on pre-compiled high-level code. Nevertheless, PROLEAD\_SW and other leakage simulators (except ASCOLD) are not exhaustive, i.e. PROLEAD\_SW does not check every possible input what allows a small false-negative probability. However, PROLEAD\_SW reports its accuracy, i.e. probability for false positives and false negatives. PROLEAD\_SW is based on our newly introduced generic leakage model for CPUs and might not include every possible leakage. Thus PROLEAD\_SW is not exhaustive in terms of possible leakages and not exhaustive in terms of simulated inputs.

### 1.3 Outline

In Section 2, we introduce the basic concepts of Boolean masking and the probing security model. Moreover, we highlight the most relevant characteristics of the Advanced Risc Machines (ARM) architecture. Based on this, we present the original PROLEAD framework and the applied ARM emulator, called M-Ulator, in Section 3. Next in Section 4, we present the workflow of PROLEAD\_SW while we focus on covering micro-architectural leakages. Finally, we evaluate multiple masked software implementations in Section 5, and discuss the results and the limitations of PROLEAD\_SW. We finally conclude the paper in Section 7.

## 2 Background

### 2.1 Notations

We denote single elements with lower-case letters, e.g. a single-bit value  $v \in \mathbb{F}_2$ , a single instruction  $i$ , a single register  $r$ , or a single probe  $p$ . For a set of elements we use bold upper case letters, e.g.  $\mathbf{V}$  denotes a set of values,  $\mathbf{I}$  denotes a set of instructions,  $\mathbf{R}$  denotes a set of registers, and  $\mathbf{P}$  denotes a set of probes. A single element inside a set can be indicated by its corresponding index, e.g.  $v_0 \in \mathbf{V}$  denotes the first value in  $\mathbf{V}$ . We distinguish between sets where the order of their elements matters, e.g.  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$ , and sets where the order of their elements does not matter, e.g.  $\mathbf{P} = \{p_0, \dots, p_{|\mathbf{P}|-1}\}$ . Further, we denote sets containing all elements of a specific type with calligraphic font, e.g.  $\mathcal{R}$  denotes a set with all registers and  $\mathcal{P}$  denotes a set of all possible probes. Finally, we denote functions of a particular software implementation with serif fonts, e.g. `main` or `cipher`.

### 2.2 Boolean Masking

Masking [CJRR99] is a common and well-studied approach to protect cryptographic implementations against SCA attacks. In *d-order Boolean masking*, a set of  $d + 1 \geq 2$  independently and uniformly distributed *shares*  $\{x^0, \dots, x^d\} \in \mathbb{F}_2^{d+1}$  represents a sensitive variable  $x \in \mathbb{F}_2$  such that  $x = \bigoplus_{i=0}^d x^i$ . To get the sharing of  $x$ , we can choose the shares in  $\{x^0, \dots, x^{d-1}\} \in \mathbb{F}_2^d$  uniformly at random and compute  $x^d$  as  $x^d = \left( \bigoplus_{i=0}^{d-1} x^i \right) \oplus x$ . To avoid any leakage revealing information about  $x$ , all executed instructions process  $\{x^0, \dots, x^d\}$ , i.e., the shared (and randomized) representation of  $x$ , instead of  $x$  itself.

## 2.3 Probing Security

The simple and abstract *d-probing model* [ISW03] formalizes a *d-probing adversary* by gaining access to up to  $d$  values  $\{x_0, \dots, x_{d-1}\}$  via placing probes on internal wires of the circuit. Hence, an implementation is *d-probing secure* if no *d-probing adversary* can extract any sensitive information from  $\{x_0, \dots, x_{d-1}\}$ . Nevertheless, even probing secure implementations often fail to achieve the desired level of security in practice due to unintentional physical effects inside the circuit (e.g. CPU). Physical defaults such as *glitches* and *transitions* have been identified as the main reasons for the security degradation of a practical masking scheme running on a hardware platform. To consider physical defaults during the security evaluation, the *robust d-probing model* [FGP<sup>+</sup>18] extends the basic probing model.

**Glitches.** As hardware circuits are not ideal, imbalanced routing and switching delays inside the CPU may lead to input signals which arrive asynchronously at their corresponding gate. Such timing differences can result in a *glitch*, i.e., an unexpected but temporary output change before the output signal reaches its intended state. The robust probing model covers glitches by allowing an adversary to place *glitch-extended probes* on arbitrary wires of the circuit.

**Definition 1** (Glitch-extended Probe). A glitch-extended probe on wire  $w$  models the impact of glitches by recording all stable signals that contribute to  $w$ .

**Transitions.** If an instruction changes a wire or register state, the resulting *transitions* may reveal information about the old and the new values based on the number of changed bits during the overwrite. To model transitions in the robust probing model, an adversary gains the ability to place *transition-extended probes* on arbitrary wires of the circuit.

**Definition 2** (Transition-extended Probe). A transition-extended probe on wire  $w$  models the impact of transitions by recording two consecutive values carried by  $w$ .

Consequently, the effects of glitches and transitions during the evaluation of masked software depend on the underlying hardware, i.e. the micro-architecture of the processor. Hence, to model the leakage correctly, the processor’s hardware design must be considered. The first step in this direction is the adaption of non-completeness to cover at least a subset of micro-architectural effects. Gaspoz et al. [GD22] summarized their observations in the following lemmas.

**Lemma 1.** *Software storing all shares of a secret bit within the same register may leak.*

**Lemma 2.** *Software storing all shares of a secret bit on the same register index may leak.*

However, the practical leakage of masked software strongly depends on the (often secret) design of the CPU. Moreover, fulfilling **Lemma 1** and **Lemma 2** does only protect against micro-architectural issues considered in [GD22] while every other micro-architectural effect needs additional consideration.

## 2.4 Advanced Risc Machines (ARM)

In this work, we target Cortex-M processors [Yiu16]. Compared to other ARM processor families, Cortex-M processors are efficient in terms of area, energy, and price. Today, different Cortex-M processors, including low-energy ones, such as the M0+, up to high-end ones, such as Cortex-M7, exist on the market. Hence, the designers can choose the most fitting CPU based on their particular needs. The large variety of Cortex-M processors leads to a wide range of possible applications, reaching from Internet of Things (IoT)

devices for consumers to automated industrial applications. In this work, we focus on the ARMv6-M, ARMv7-M, and ARMv7E-M Instruction Set Architecture (ISA)s that allow us to evaluate implementations for Cortex-M0 up to the Cortex-M7 processors. All these CPUs are common in having fifteen 32-bit general purpose registers  $\{r_0, \dots, r_{14}\}$ , the Program Counter (PC) ( $r_{15}$ ) and the Program Status Register (PSR) ( $r_{16}$ ). We refer to the list of all registers with  $\mathcal{R} = \{r_0, \dots, r_{16}\}$ . The ARMv6-M architecture implements a comparably small instruction set, resulting in a simple processor design. However, this ISA limits the effective register usage to the lower eight registers. The instruction set of the ARMv7-M allows more complex operations and supports advanced data processing and bit field manipulations. Finally, the ARMv7E-M architectures extend the ARMv7-M ISA by Digital Signal Processing (DSP) operations. As the instruction set of a higher architecture is a superset of a lower one, Cortex-M binaries provide upward compatibility.

## 3 Related Works

### 3.1 PROLEAD

PROLEAD [MM22] is a probing-based hardware leakage detection tool combining the benefits of formal verification and leakage simulation. It evaluates the robust-probing security of a given gate-level netlist of the circuit under test by simulating the underlying logic. Its abilities include the consideration of glitches and transitions as well as univariate and multivariate adversaries at arbitrary security orders. Internally, PROLEAD considers every possible robust-probing adversary by generating a list of all possible probing sets observed by an attacker. By simulating the netlist, PROLEAD estimates the distributions of each probing set under different input settings. Finally, it checks the independence of distributions based on a statistical hypothesis test. If a significant dependency is detected for at least one probing set, PROLEAD reports a general information leakage and details of the most leaking probing set.

**Efficiency.** Publicly available benchmarks show that PROLEAD can easily handle first-order masked cipher hardware cores in minutes to hours. However, naturally increasing the security order increases the runtime of PROLEAD exponentially. While evaluating second-order masked cipher cores is still possible, the authors of PROLEAD only evaluated a single third-order masked AES S-box. PROLEAD’s runtime and the amount of necessary RAM space strongly depends on the probing set size, i.e. the number of considered probes in a probing set while the number of possible adversaries, i.e. the number of probing sets, plays a subordinate role.

**Reliability.** For each test result, PROLEAD calculates the statistical power of the underlying test procedure to evaluate the result’s reliability. Formally, PROLEAD estimates the false-positive probability  $p$  and false-negative probability  $\beta$  based on a fixed effect size  $\varphi$ . Hence, if PROLEAD detects leakage, it informs the user about the probability that the leakage is a false-positive result, and that the design might be secure. By default, PROLEAD reports leakage with a false-positive probability of  $p < 10^{-5}$ . Moreover, if PROLEAD detects no leakage, it informs the user about the probability that the result is a false negative, i.e. that no leakage with an effect size  $\varphi$  was detected. By default, PROLEAD reports a design as secure with a false-negative probability of  $\beta < 10^{-5}$  and  $\varphi = 0.1$ . All parameters can be adjusted by the user, e.g. to detect smaller leakages (by decreasing  $\varphi$ ) or to increase the reliability of the results (by decreasing  $p$  and  $\beta$ ). We remark that the number of simulations has a decisive influence on reliability. That is why PROLEAD continuously monitors the remaining number of simulations required to achieve the specified parameters.

### 3.2 M-Ulator

To execute ARM binaries on non-ARM processors we need an emulator that replicates the environment by providing the same behavior as the target platform. For PROLEAD\_SW we have chosen M-Ulator introduced in [HSP21] and publicly available on GitHub<sup>3</sup>. The tool was originally written for fault simulation on ARMv6-M and ARMv7-M binaries and implements its functionality based on publicly available architecture specifications. M-Ulator is not bounded to a specific processor but rather supports every core running the provided ISA. As it is not dedicated to a particular processor, it allows only instruction-accurate simulation. However, this permits a wider range of applications.

The advantages of this emulator are manifold. Unlike graphical emulators such as Visual<sup>4</sup>, we can observe intermediates as well as their changes in registers and in the memory at every single stage of the execution. As we need to directly probe the simulated intermediates, having a detailed insight is a necessary condition. Additionally, an emulator that can run the compiled machine code gives us a more realistic behavior than evaluating high-level descriptions. As our probing-based approach requires computation-intense tasks the emulator should be preferably as fast as possible. According to [HSP21], M-Ulator outperforms state-of-the-art emulators in terms of speed, e.g., compared to the QEMU-based Unicorn<sup>5</sup>.

## 4 Technique

We model each software, i.e., a compiled binary, as an ordered set of instructions  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  while each instruction executes its mnemonic  $m$  on a given set of operands. For the operands, we differentiate between source locations  $s \in \mathbf{S} \subseteq \mathcal{S}$ , i.e., memory locations to read, and destination locations  $d \in \mathbf{D} \subseteq \mathcal{D}$ , i.e., memory locations to write. Here, we denote  $\mathcal{S}$  (resp.  $\mathcal{D}$ ) as the set of all possible sources (resp. destinations). Both can be either registers or a memory cell of the RAM while hard-coded constants are possible sources as well. To distinguish locations, we define a `type` function that outputs `type(s) = reg` if  $s$  is a register index, `type(s) = mem` if  $s$  is a RAM address, and `type(s) = const` if  $s$  is a constant value. Consequently, we formalize an instruction  $i \in \mathbf{I}$  as a triple  $i = (m, \mathbf{S}, \mathbf{D})$  storing the instruction's mnemonic, a list of sources, and a list of destinations.

### 4.1 CPU-Independent Modelling of Micro-Architectural Effects

As software abstracts the underlying hardware's behavior, applying the robust  $d$ -probing model on the CPU design while executing software would reduce the problem of verifying software to the formal verification of hardware. While this would allow the verification under the specific CPU design we argue it would be beneficial for software developers to get a broader verification statement that is independent of a concrete realization of a CPU. Instead of evaluating the security of the software under multiple different CPU designs where each verification only holds for the specific netlist, the verification under one generic model accelerates the verification and allows a generalization of the security statement. Further, the concrete CPU design, e.g. for an ARM CPU, is most likely not available, which makes the hardware's formal verification impossible. While ARM provides hardware netlist of certain cores under the Arm Academic Access model<sup>6</sup>, its acquisition is limited to academic institutions, which makes the hardware's formal verification for the general public difficult.

<sup>3</sup><https://github.com/emsec/arm-fault-simulator>

<sup>4</sup><https://salmanarif.bitbucket.io/visual/index.html>

<sup>5</sup><https://www.unicorn-engine.org/>

<sup>6</sup><https://www.arm.com/resources/research/enablement/academic-access>



Initially, we define abstract probes for evaluating software in line with the  $d$ -probing model. To model probes on arbitrary locations of the RAM, we extend  $\mathcal{R}$  by an imaginary register  $r_{17}$  representing the currently targeted position in RAM. We denote the extended register list as  $\mathcal{R}^*$ .

**Definition 3** (Standard Probe). A standard probe  $p = (i, r, b)$  with  $i \in \mathbf{I}$ ,  $r \in \mathcal{R}^*$ , and  $b < 32$  is placed on the bit  $b$  of register  $r$  during instruction  $i$ .

**Definition 4** (Transition-extended Probe). A transition-extended probe  $p = (\langle i, i' \rangle, r, b)$  with  $i, i' \in \mathbf{I}$ ,  $r \in \mathcal{R}^*$ , and  $b < 32$  records bit  $b$  of register  $r$  during instructions  $i$  and  $i'$ . It holds that  $i'$  denotes another instruction executed before executing  $i$ .

**Definition 5** (Probing Set). A probing set  $\mathbf{P} = \{p_0, \dots, p_{|\mathbf{P}|-1}\}$  defines a set of probes.

From Lemma 1 and Lemma 2 we derive horizontal and vertical probes as a new class of probes for modeling micro-architectural effects. However, both are in line with robust-probing security as they are the possible outcome of a single glitch-extended probe. We remark that both definitions describe extended probes. Hence, horizontal and vertical probes expand to an equivalent probing set containing standard probes.

**Definition 6** (Horizontal Probe). A horizontal probe  $p_h = (i, r, \mathbf{B})$  with  $i \in \mathbf{I}$ ,  $r \in \mathcal{R}^*$ , and  $\mathbf{B} \subseteq \llbracket 0, 32 \rrbracket$  records all bits  $b \in \mathbf{B}$  of register  $r$  during instruction  $i$ .

**Definition 7** (Vertical Probe). A vertical probe  $p_v = (i, \mathbf{R}, b)$  with  $i \in \mathbf{I}$ ,  $\mathbf{R} \subseteq \mathcal{R}$ , and  $b < 32$  records bit  $b$  of all registers  $r \in \mathbf{R}$  during instruction  $i$ .

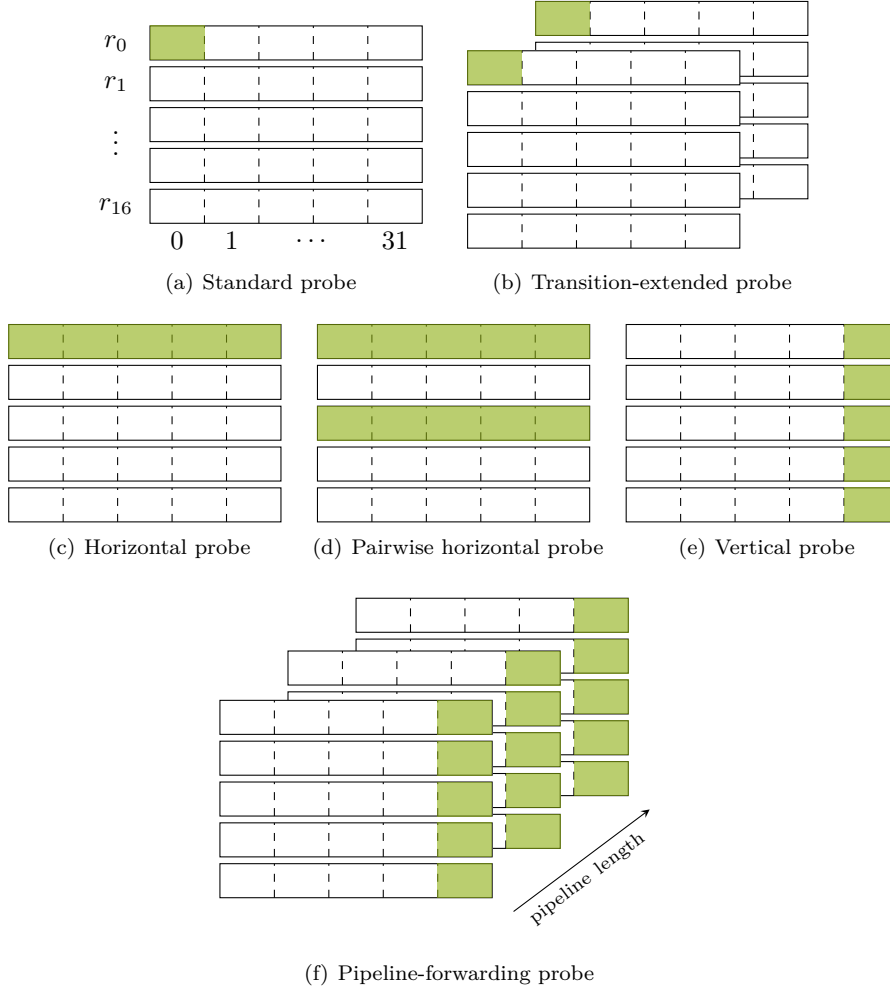
#### 4.1.1 Covered Micro-Architectural Effects

We remark that Definition 6 and Definition 7 allow the verification under fine-grained leakage models if we match  $R$  and  $B$  to a concrete CPU. However, CPU designs are usually missing and our goal is to have a CPU-independent verification. In order to cover all possible occurrences of the considered micro-architectural effects we set  $\mathbf{R} = \mathcal{R}^*$  and  $\mathbf{B} = \llbracket 0, 31 \rrbracket$ . Hence, every horizontal probe records *all* bits of a register, and every vertical probe records bits with the same index in *all* registers. Next, we discuss various micro-architectural leakage sources and show how we model them with horizontal and vertical probes.

**Neighbor Leakage Effect (NLE).** Whenever the CPU reads or writes a register, a multiplexer tree guides the value from the register to the ALU or pipeline and vice-versa. Therefore, the multiplexer tree connects multiple registers and decides the forwarded register state based on a select signal, i.e., the register or memory address. We remark that, for example, the Cortex-M3 implements this particular scheme [CGD18] along with other CPUs [GHP<sup>+</sup>21] (see Figure 2). The resulting leakage is commonly referred to as NLE [PV17]. As the multiplexer tree connects bits with the same index from multiple registers, placing a glitch-extended probe on one multiplexer-tree output bit cascades to probes on multiple registers. Hence, this effect confirms Lemma 2 and indicates that vertical probes model the NLE.

**Proposition 1** (NLE-Security). *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides NLEs-security, iff no vertical probe  $p_v = (i, \mathcal{R}, b)$  with  $i \in \mathbf{I}$  and  $b < 32$  does reveal any sensitive information.*

**Bit-wise Interaction Leakage (BIL).** While executing an instruction, the ALU may combine different bits of an operand register through combinational logic [GD22]. In particular, combining multiple register bits is necessary to perform arithmetic operations, e.g., ADD or SUB. Therefore, placing a glitch-extended probe on one ALU output results



**Figure 1:** Different abstract probes on the registers.

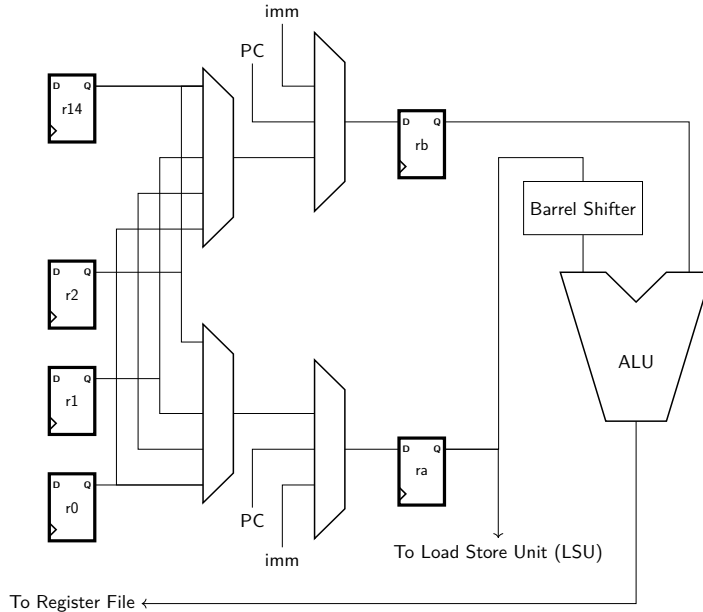
in probes on multiple bits of the operand register. Hence, this effect confirms Lemma 1 and indicates that horizontal probes model BIL. As the ALU processes two operands, the BIL is not bounded to one register but to bits within both source registers of an instruction. To model BIL for a particular register pair, we define pairwise horizontal probes.

**Definition 8** (Pairwise Horizontal Probe). A pairwise horizontal probe  $p_h = (i, \{r_0, r_1\}, \mathbf{B})$  with  $i \in \mathbf{I}$ ,  $r_0, r_1 \in \mathcal{R}$ , and  $\mathbf{B} \subseteq \llbracket 0, 31 \rrbracket$  records all bits  $b \in \mathbf{B}$  of registers  $r_0$  and  $r_1$  during instruction  $i$ .

We place pairwise horizontal probes on all register tuples in every instruction  $i \in \mathbf{I}$ . However, as only the value of the destination register changes during an instruction, the tuples encompassing the destination register are enough. If we add pairwise horizontal probes with the destination register for every instruction, we cover all register combinations.

**Proposition 2** (BIL-Security). *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides BILs-security, iff no pairwise horizontal probe  $p_h = (i, \{r_0, r_1\}, \llbracket 0, 31 \rrbracket)$  with  $i \in \mathbf{I}$  and  $r_0, r_1 \in \mathcal{R}$  does reveal any sensitive information.*

As  $r_{17}$  does not connect to the multiplexer tree we ignore  $r_{17}$  while building pairwise horizontal probes, but we place an additional horizontal probe on  $r_{17}$  if  $i$  decodes an



**Figure 2:** Simplified ARM Cortex-M3 pipeline structure [CGD18].

interaction with the RAM to model the multiplexers between  $r_{17}$  and the registers, as shown in [MPW22].

**Memory Overwrite Effect (MOE).** If an instruction overwrites a storage element, the corresponding transition leaks information about the old and new states. To abstract transitional leakage, we can apply transition-extended probes as defined in Definition 4. Consequently, every aforementioned probe must become transition-extended in order to capture all possible MOEs. Further, we cover MOEs on the RAM by placing transition-extended probes on every targeted address of the RAM.

**Proposition 3 (MOE-Security).** *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides MOEs-security, iff for every instruction  $i \in \mathbf{I}$  that modifies  $s$ , i.e. a register in  $\mathcal{R}^*$  or a particular address of the RAM, no transition-extended probe  $p = \langle (i, i'), s, b \rangle$  with  $i' \in \mathbf{I}$  and  $b < 32$  does reveal any sensitive information while  $i'$  denotes the last instruction that modified  $s$  before executing  $i$ .*

**Memory Remnant Effect (MRE).** While MOEs lead to transitions on the same destination register, it was shown in [PV17] and experimentally verified in [MPW22] that consecutive memory accesses lead to leakage even if the source and destination registers are different. One explanation is that a hidden (shadow) register caches values for any RAM activities (either writing to or reading from RAM) or that the related data bus acts as a hidden register [MMT20]. Hence, transitions on the hidden register cause the MRE. Moreover, the results in [MPW22] regarding memory bus width indicate that the hidden register has a size of 32 bits. Hence, instructions interacting with the RAM to load/store only 16 or 8 bits lead to the same transition similar to the related instruction processing a 32-bit word while a multiplexer tree forwards particular byte(s) to the corresponding register.

**Proposition 4 (MRE-Security).** *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides MREs-security, iff for every instruction  $i \in \mathbf{I}$  that interacts with the RAM, i.e. reading the new state of  $r \in \mathcal{R}$*

from RAM or writing  $r$  to the RAM, no transition-extended probe  $p = (\langle i, i' \rangle, r_{17}, b)$  with  $i' \in \mathbf{I}$  and  $b < 32$  and no horizontal probe  $p_h = (i, r_{17}, \llbracket 0, 31 \rrbracket)$  does reveal any sensitive information while  $i'$  denotes the last instruction that interacted with the RAM before executing  $i$ .

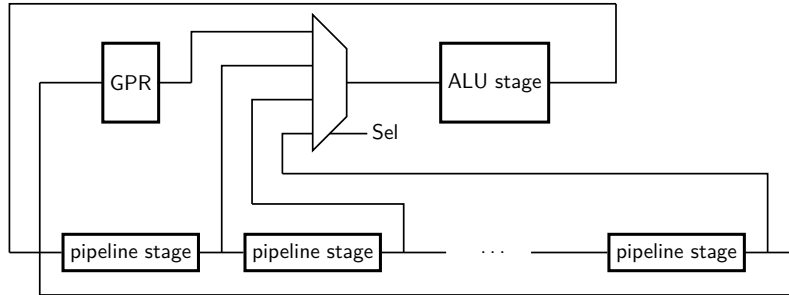
This proposition corresponds to transition-extended probes on all bits of  $r_{17}$  for every instruction that accesses the RAM. Moreover, the results in [MPW22] revealed combined leakage from different bits of  $r_{17}$  highlighting the need for a horizontal probe on  $r_{17}$ .

**Pipeline Register Overwrite (PRO).** If the CPU implements a pipeline, the integrated pipeline registers are prone to transitions combining operands of two consecutive instructions. Again, the Cortex-M3 implements a transition-prone pipeline (cf. Figure 2) [PV17]. Moreover, the pipeline effects of various ARM CPUs has been shown through practical experiments in [MPW22]. According to the robust probing model, we consider transitions in the pipeline by probing consecutive instructions.

**Proposition 5 (PRO-Security).** *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides PROs-security, iff no transition-extended probe  $p = (\langle i, i' \rangle, r, b)$  with  $i, i' \in \mathbf{I}$ ,  $r \in \mathcal{R}$ , and  $b < 32$  does reveal any sensitive information while  $i'$  denotes the last instruction that was executed right before  $i$ .*

PROs are already covered by vertical probes that are used for NLE security as both values (old and new) inside the pipeline are stored in their original registers as well.

**Pipeline Forwarding Effect (PFE).** To reduce pipeline stalls in case of data dependencies forwarding logic allows to access values from later pipeline stages without waiting for the values to be written to the destination registers. Gigerl et al. [GPM21] demonstrate on the RISC-V SweRV core that such logic is able to pose a severe security threat as it potentially allows to observe values of multiple pipeline registers within a single probe. Let us consider the schematic representation of a CPU with its pipeline stages in Figure 3.

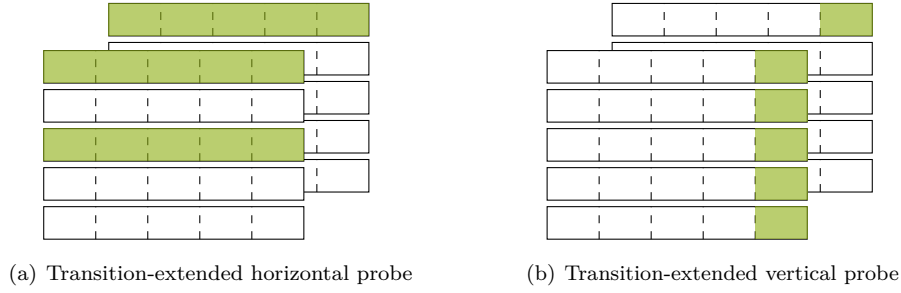


**Figure 3:** Schematic depiction of pipeline stages and forwarding logic.

As Gigerl et al. point out, due to occurring glitches probing the output of the combinatorial forwarding logic, i.e., the output of the multiplexer in Figure 3, could give adversaries access to its inputs and thus to multiple values within the pipeline. Furthermore, Grandmaison et al. [dGHM22] report that such a forwarding mechanism also exists for ARM processors.

**Definition 9 (Pipeline Forwarding Probe).** A pipeline forwarding probe  $p_f = (\langle i, i-1, \dots, i-x \rangle, \mathcal{R}, b)$  with  $x$  being the pipeline length,  $i, i-1, \dots, i-x \in \mathbf{I}$ , and  $b < 32$  records bit  $b$  of all registers  $r \in \mathcal{R}$  during a window of  $x$  instructions.

**Proposition 6 (PFE-Security).** *Software  $\mathbf{I} = \langle i_0, \dots, i_{|\mathbf{I}|-1} \rangle$  provides PFEs-security in a pipeline containing  $x \in \mathbb{N}$  pipeline stages, iff no pipeline-forwarding probe  $p = (\langle i, i-1, \dots, i-x \rangle, \mathcal{R}, b)$  with  $i, i-1, \dots, i-x \in \mathbf{I}$  and  $b < 32$  does reveal any sensitive information.*



**Figure 4:** Different transition-extended probes on the registers.

#### 4.1.2 Combined Occurrence of Glitches and Transitions

In contrast to [GD22], we consider glitches and transitions simultaneously and discuss their combined leakage. In particular, the aforementioned models can be freely combined in order to consider the combined occurrences of micro-architectural effects. Therefore, we enable to extend all glitch-extended probes, i.e. horizontal and vertical probes, by exchanging all resulting standard probes with transition-extended probes. In particular, if the state of a register  $r$  changes during instruction  $i$  all probes  $p = (i, r, b)$  become transition-extended probes  $p' = (\langle i, i' \rangle, r, b)$  while  $i'$  denotes the last instruction that changes the state of  $r$  before  $i$ . For registers that keep their value during instruction execution, no transition extension is needed. For horizontal and vertical probes, we define their transition-extended variants as follows and visualize the concept in Figure 4.

**Definition 10** ((Pairwise) Transition-extended Horizontal Probe). A transition-extended horizontal probe  $p_h = (\langle i, i' \rangle, \{r, r'\}, \mathbf{B})$  with  $i, i' \in \mathbf{I}$ ,  $r, r' \in \mathcal{R}$ , and  $\mathbf{B} \subseteq \llbracket 0, 31 \rrbracket$  records all bits  $b \in B$  of registers  $r$  and  $r'$  during instruction  $i$  and all bits  $b \in \mathbf{B}$  of register  $r$  during instruction  $i'$ , while  $i'$  denotes the last instruction that modified  $r$  before executing  $i$ .

Definition 10 denotes  $r$  as destination register of  $i$ . Hence, all probes on  $r$  during  $i$  become transition-extended, while for probes on  $r'$ , no transition-extension happens.

**Definition 11** (Transition-extended Vertical Probe). A transition-extended vertical probe  $p_v = (\langle i, i' \rangle, \mathbf{R}, b)$  with  $i, i' \in \mathbf{I}$ ,  $\mathbf{R} \subseteq \mathcal{R}$ , and  $b < 32$  records the  $b$ -th bit of all registers in  $\mathbf{R} \subseteq \mathcal{R}$  during instruction  $i$  and the  $b$ -th bit of register  $r \in \mathbf{R}$  during instruction  $i'$ , while  $i'$  denotes the last instruction that modified  $r$  before executing  $i$ .

Again,  $r$  is the destination register of  $i$  while bit  $b$  of  $r$  is the only bit changing in the vertical probe. Hence, only  $p = (i, r, b)$  becomes transition-extended. We remark that in PROLEAD\_SW based on the settings defined by the user/evaluator, we generate all above-explained probes for every instruction.

## 4.2 Input Files

**Configuration.** First, PROLEAD\_SW receives all evaluation settings in form of a user-defined configuration file. In particular, the configuration file contains the following settings.

- Compiler options, such as the compiler flags to use, e.g. level of optimization.
- Simulation settings, such as the primary inputs given to M-Ulator, the number of simulations, and the number of instructions to evaluate.
- Evaluation settings, such as statistical parameters, physical defaults to consider, and the underlying security order.

**Source Code.** Second, the user has to provide the source code, written in C or ARM assembly. To increase accuracy, we try to minimize the gap between the provided source code and the final software running on a CPU. Therefore, we reduce the necessary constraints to a minimum. The remaining restrictions, which are commonly present in source files running on a CPU anyway, are the following.

- As PROLEAD\_SW needs a starting point to emulate, we pre-empt the presence of a `main` function.
- Moreover, the source files must contain a `cipher` function encompassing the sequence to evaluate. Instructions outside of `cipher`, i.e., within `main` but not inside `cipher`, are not part of the evaluation. This is reasonable as it allows the user to do pre-processing and post-processing, e.g., (un-)masking with a dedicated algorithm which would naturally cause leakage when this instruction sequence is probed.
- The user has to define global variables with the associated names of the primary inputs in the configuration file. This creates a link for PROLEAD\_SW between the configuration and source files and ensures that the inputs will have a dedicated range in memory. Otherwise, we cannot guarantee that writing the inputs into memory does not interfere with or overwrite other elements, e.g. global arrays that were defined in the source file.

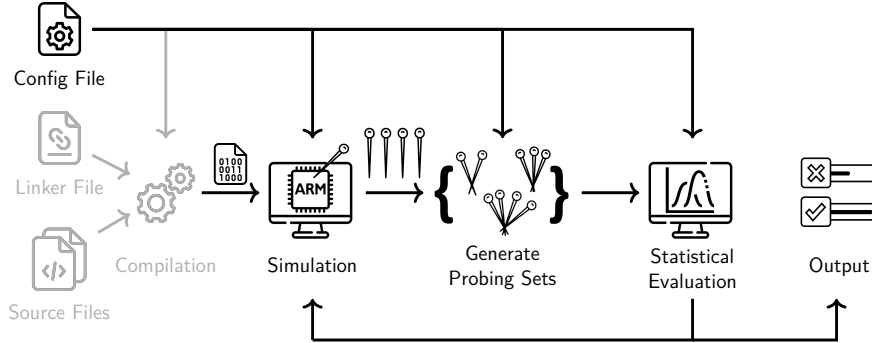
**Linker.** Third, a linker script is necessary to describe how sections, e.g., RAM or Flash, should be mapped, thus controlling the memory layout. To find the global input’s memory address, we require the existence of the `_edata` symbol within the linker script to denote the end of the data section. Additionally, the user can introduce a four-byte `.randomness` section in memory containing a single 32-bit random value, which gets refreshed whenever accessed. We recommend using the given randomness to seed user-provided Pseudo-Random Number Generator (PRNG) implementations since utilizing the randomness source solely would result in a different binary running on the actual micro-controller. This is because while we have a dedicated memory area from which we load fresh randomness, micro-controllers need to sample their randomness either from a peripheral or implemented PRNG, which results in different instructions and register utilization.

**Binary.** As an alternative to the linker and source files, the user can compile the source code outside of PROLEAD\_SW and provide the compilation results, including the binary as Executable and Linking Format (ELF), the corresponding map file, and a disassembly file of the binary in text format. Note that all constraints explained above (in paragraph **Source Code**) still apply if the compilation is done outside of PROLEAD\_SW. The map file contains the address and size of every load region, execution region, and input section of the binary. PROLEAD\_SW demands this file to initialize the emulator correctly.

### 4.3 Evaluation

Figure 5 describes the high-level procedure of PROLEAD\_SW. If required, the source files and the linker file are compiled to the final binary and passed to the `M-Ulator`. It parses the required information from the output of the previous step to instantiate the emulator. Together with the information provided in the configuration file, `M-Ulator` executes one instance of the binary. During the execution, it simulates the intermediate states of the considered registers and generates all possible probes which can be placed by an adversary. Afterwards, multiple probes build different probing sets accordingly to the user’s given configuration. Further, statistical tests evaluate the independence of observations within every probing set individually. In the following, we take a deeper look at each stage

and describe its operations. We remark that we mainly adjust the simulation and the generation of probes compared to the original hardware version of PROLEAD while the statistical evaluation is generic and can therefore be used for both hardware and software.



**Figure 5:** High-level overview of PROLEAD\_SW.

## 4.4 Compilation

If the user provides only the source files and corresponding linker file, PROLEAD\_SW itself invokes the compiler with the specified compiler flags from the configuration file. To compile the source-code we apply the `arm-none-eabi-gcc` compiler<sup>7</sup>. The compilation step generates the required ELF binary together with the disassembly file and map file. We note that the user is free to use arbitrary compilers if it provides the required information, such as the map file, the disassembled binary, and a compiled source code for the ARMv6-M, ARMv7-M, or ARMv7E-M architecture.

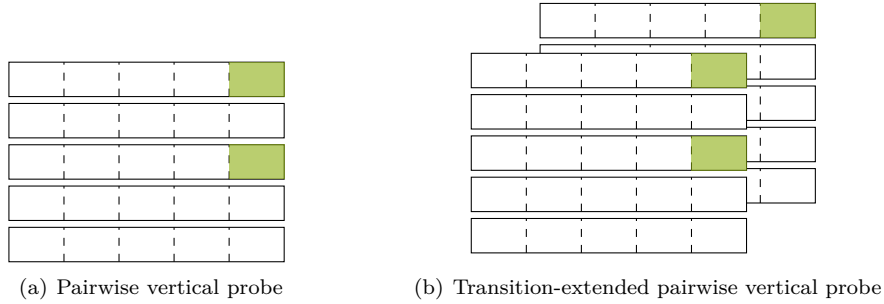
### 4.4.1 Initialization

Initially, PROLEAD\_SW allocates three empty data structures with the following properties.

- $\mathcal{P}$  is an empty set that stores all probes placed during one simulation.
- $\mathcal{G}$  is an empty set that stores the probing sets of all simulations and their corresponding distributions in form of contingency tables.
- $\mathbf{V}$  is a  $(|\mathbf{I}|, |\mathcal{R}^*|, 32)$ -matrix that stores all changes on a simulated state.

Based on the information provided in the map file, PROLEAD\_SW extracts the starting address and the size of RAM, respectively Flash, together with the memory location of the global inputs and, if available, the position of the dedicated `.randomness` section. Subsequently, M-Ulator copies the start address of `main` into the PC and the start address of the stack into the Stack Pointer (SP). Next, PROLEAD\_SW randomly selects one of the user-defined groups, which describes the values of the inputs for the current simulation. For example, in a fixed-vs-random setting with an input size of 32 bits, the user specifies the two groups `32'hda39a3ee` and `32'h$$$$$$$$` in the configuration file where `32'hda39a3ee` denotes a fixed input and `32'h$$$$$$$$` a random vector. If the configuration denotes certain inputs to be presented in a Boolean shared form, PROLEAD\_SW masks those inputs automatically, i.e., generates a fresh sharing of every corresponding bit at the start of every simulation. Afterwards, the inputs will be placed in the dedicated memory regions reserved for the global inputs from the source file. The last step of the initialization phase simulates the binary until the beginning of the `cipher` function, where the actual probe generation starts.

<sup>7</sup><https://github.com/marketplace/actions/arm-none-eabi-gcc-gnu-arm-embedded-toolchain>



**Figure 6:** Pairwise vertical probes to improve efficiency.

## 4.5 Simulation

To support a broader range of binaries, we extended the capabilities of `M-Ulator` to handle the ARMv7E-M ISA, which enables the use of DSP instructions commonly used in post-quantum schemes. In the following, we shortly explain the general procedure during the simulation of a single instruction  $i \in \mathbf{I}$ . For more detailed explanations, we refer to the original `M-Ulator` paper [HSP21]. First, `M-Ulator` fetches the PC register value. Afterwards, it computes the memory address of  $i$ , loads the encoded instruction from memory, and passes it to the decoder. The decoder translates the opcode into a machine-readable operation. The execution unit performs the actual logic of the operation and sets all internal modifications, such as status flags, according to the specification. After the execution, we store the content of all registers that changed during instruction  $i$  in the matrix  $\mathbf{V}$ . Particularly, we go through all destination locations  $d \in \mathbf{D}$  and store each bit at bit-position  $b$  at position  $(i, d, b)$  in  $\mathbf{V}$ , i.e.,  $v_{i,d,b}$ . To be able to quickly identify the instruction  $i' \in \mathbf{I}$  that contributed to the previous change of register  $r \in \mathcal{R}^*$  we extend `M-Ulator` further by storing the previous instruction  $i'$  that changed the value of the register  $r$ .

### 4.5.1 Generation of Probes

If an instruction  $i$  is part of the `cipher` function, we place sufficient probes to detect possible leakages in  $i$ . Additionally, we place subsets of each vertical probe, so-called pairwise vertical probes as visualized in Figure 6. We would like to remark that the pairwise vertical probes are already covered by complete vertical probes. However, pairwise modeling leads to smaller probing sets which are more efficient to evaluate. Moreover, we expect to detect pairwise leakages faster than with vertical probes, i.e. by fewer simulations. Nevertheless, depending on the ALU design, if glitches reveal information about more than two registers at the same time, the vertical probes must be considered. This is among the configurations of `PROLEAD_SW` which can be set by the user.

We formalize our procedure in Algorithm 1. In Line 3, `PROLEAD_SW` checks whether the result of  $i$  is stored in a register or RAM. If  $i$  writes its result in a register, we create standard probes on all bits of the destination register  $r$ , i.e.,  $p_0 = (i, r, 0), \dots, p_{31} = (i, r, 31)$  as done in Line 15. If enabled, `PROLEAD_SW` will further create horizontal and vertical probes. Following Definition 6 and Definition 7, an additional horizontal probe  $p_h = (i, r, \llbracket 0, 31 \rrbracket)$  (cf. Line 6) jointly probes all bits of  $r$  while an additional vertical probe  $p_v = (i, \mathcal{R}, b)$  (cf. Line 18) probes the  $b$ -th bit of all registers. However, reading or writing data from or to RAM invokes the hidden memory register  $r_{17}$ . Hence, if  $i$  interacts with the RAM, as checked in Line 11, we instantiate additional standard probes on  $r_{17}$ . Finally,  $\mathcal{P}$  stores all probes of the current simulation.



**Algorithm 1** Probe Generation

---

**Input:**  $i = \langle m, \mathbf{S}, \mathbf{D} \rangle$  ▷ An instruction considered for evaluation  
**Input:**  $\mathcal{P}$  ▷ Set of probes  
**Input:**  $\mathcal{R}$  ▷ A list of all registers  
**Output:**  $\mathcal{P}$  ▷ Updated set of probes according to the execution of  $i$

- 1:  $\mathbf{D}^* \leftarrow \emptyset$
- 2: **for**  $\forall d \in \mathbf{D}$  **do**
- 3:   **if**  $\text{type}(d) = \text{reg}$  **then**
- 4:      $r \leftarrow d$  ▷ The corresponding target register
- 5:      $\mathbf{D}^* \leftarrow \mathbf{D}^* \cup \{r\}$
- 6:      $p_h \leftarrow (i, r, \llbracket 0, 31 \rrbracket)$  ▷ Place a complete horizontal probe on register  $r$
- 7:      $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_h\}$
- 8:     **for**  $\forall r' \in \mathcal{R} \setminus r$  **do**
- 9:        $p_h \leftarrow (i, \{r, r'\}, \llbracket 0, 31 \rrbracket)$  ▷ Place a pairwise horizontal probe on register  $r$
- 10:        $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_h\}$
- 11:   **if**  $\text{type}(d) = \text{mem} \vee \exists s \in S \text{ s.t. } \text{type}(s) = \text{mem}$  **then**
- 12:      $\mathbf{D}^* \leftarrow \mathbf{D}^* \cup \{r_{17}\}$  ▷ Add the hidden memory register
- 13: **for**  $\forall r \in \mathbf{D}^*$  **do**
- 14:   **for**  $\forall b \in \llbracket 0, 31 \rrbracket$  **do**
- 15:      $p \leftarrow (i, r, b)$  ▷ Place a standard probe on bit  $b$  of register  $r$
- 16:      $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$
- 17:     **if**  $r \neq r_{17}$  **then**
- 18:        $p_v \leftarrow (i, \mathcal{R}, b)$  ▷ Place a complete vertical probe on bit  $b$
- 19:        $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_v\}$
- 20:       **for**  $\forall r' \in \mathcal{R} \setminus r$  **do**
- 21:          $p_v \leftarrow (i, \{r, r'\}, b)$  ▷ Place a pairwise vertical probe on bit  $b$
- 22:          $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_v\}$
- 23: **for**  $\forall b \in \llbracket 0, 31 \rrbracket$  **do**
- 24:    $p_f \leftarrow (\{i, i-1, \dots, i-x\}, \mathcal{R}, b)$  ▷ Place a pipeline forwarding probe on bit  $b$
- 25:    $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_f\}$

---

## 4.6 Generation of Probing Sets

According to [Algorithm 1](#), every simulation of the binary results in a set  $\mathcal{P}$  encompassing all relevant probes for evaluation. To reflect the capabilities of a  $d$ -probing adversary, we combine the individual probes to all possible  $d$ -probing sets, i.e., the information a  $d$ -probing adversary gains. We store all probing sets belonging to a certain simulated execution in the set  $\mathcal{P}^*$ . PROLEAD already implements a method to combine probes into probing sets (see. [Algorithm 2](#) of [\[MM22\]](#)) which we also apply. On the one hand, a univariate attacker places  $d$  probes during the same instruction  $i \in \mathbf{I}$ . Hence, we construct  $d$ -combinations of probes  $p \in \mathcal{P}$  that were generated during every instruction  $i$  individually and add them to  $\mathcal{P}^*$ . On the other hand, a multivariate attacker can place probes on arbitrary instructions in  $\mathbf{I}$ . In such a case, we need to allow a combination of every  $d$  arbitrarily probes in  $\mathcal{P}$  and add them to  $\mathcal{P}^*$ . Contrary to the hardware evaluation, we cannot precompute the probing sets for software as the evaluated software may not achieve constant time. In such a case, the generated probing sets may differ between multiple simulations. Therefore, we create new probing sets for every simulation and insert them into the global set  $\mathcal{G}$ . In particular, for every probing set  $\mathbf{P}$  in  $\mathcal{P}^*$ , we check whether  $\mathbf{P}$  already exists in  $\mathcal{G}$ . If so, we update the distribution of the corresponding entry in  $\mathcal{G}$  based on the simulated values observed by  $\mathbf{P}$ . Otherwise,  $\mathbf{P}$  is inserted into  $\mathcal{G}$ .

## 4.7 Probe Extension

During the insertion of a new probing set  $\mathbf{P}$  into  $\mathcal{G}$  we have to extend every probe in  $\mathbf{P}$  according to [FGP<sup>+</sup>18], i.e., we transfer all probes to standard probes. This is necessary to cover the physical defaults and micro-architectural effects. As follows, we express the procedure for the extension of probes based on glitches and transitions separately.

**Glitch-Extension.** We remark that we apply horizontal and vertical probes to cover leakages originating from glitches. Thus, we extend all  $p_h$  and  $p_v$  by substituting them with multiple standard probes recording all single bits recorded by  $p_h$  or  $p_v$ . For example consider a horizontal probe  $p_h = (i, r_2, \llbracket 0, 31 \rrbracket)$  observing all 32 bits of register  $r_2$ . After extension, the corresponding probing set contains 32 probes  $p_0 = (i, r_2, 0), \dots, p_{31} = (i, r_2, 31)$ . The same applies to the extension of a vertical probe. Let us examine  $p_v = (i, \mathcal{R}, 5)$ , i.e., probing the 5-th bit of every  $r \in \mathcal{R}$ . The extended probing set consists of 17 probes  $p_0 = (i, r_0, 5), \dots, p_{16} = (i, r_{16}, 5)$ .

**Transition-Extension.** If we consider transitions, Algorithm 1 automatically generates transition-extended probes  $p = (\langle i, i' \rangle, r, b)$  instead of standard probes  $p = (i, r, b)$ . Hence, to extend  $p$ , we generate a tuple of standard probes  $(p, p')$  with  $p = (i, r, b)$  and  $p' = (i', r, b)$ . The same procedure applies for horizontal probes, i.e., Algorithm 1 stores  $p_h = (\langle i, i' \rangle, r, \llbracket 0, 31 \rrbracket)$  instead of  $p_h = (i, r, \llbracket 0, 31 \rrbracket)$  what corresponds to 32 tuples  $(p_h, p'_h)$  and 64 standard probes in total. For vertical probes, only transitions during instruction  $i = (m, \mathbf{S}, \mathbf{D})$ , i.e., on its destinations, are considered. Therefore, a transition-extended vertical probe  $p_v = (i, \mathbf{R}, b)$  just extends  $p_v$  with a standard probe  $p' = (i', r, b)$ . The same applies to pairwise horizontal probes.

## 4.8 Statistical Evaluation

Every probing set in  $\mathcal{G}$  stores the distribution of the associated observation for every considered group in form of a contingency table. As shown in Section 4.4.1, the user specifies the underlying test procedure by setting the number of groups and a (fixed or random) value for each group. For example, a user can decide to perform a fixed versus fixed (resp. fixed versus random) test by specifying two groups with fixed but different group values (resp. one fixed and one random group value). Especially, fixed versus random tests are commonly known in the context of SCA. Note that similar to the hardware PROLEAD, the user can define more than two groups, e.g. different fixed values for each group. After each simulation, we update the contingency tables of all probing sets in  $\mathcal{G}$  observed in the last simulation. PROLEAD\_SW quantifies the statistical independence of all groups through the G-test. For each probing set, PROLEAD\_SW returns the false-positive probability by means of a  $p$ -value and reports detectable leakage if the  $p$ -value becomes smaller than  $10^{-5}$ .

### 4.8.1 Confidence

Similar to PROLEAD, we satisfy the confidentiality of PROLEAD\_SW by estimating the test metric's statistical power. Further, we rely on the same parameters, i.e., a false-positive and false-negative probability threshold of  $10^{-5}$  and an effect size of  $\phi = 0.1$ . To compute the required number of simulations to reach the given parameters, we apply the same numerical estimation as PROLEAD. For an extensive explanation of the methodology, we refer to the original PROLEAD paper [MM22]. However, we point out the most important difference between the confidentiality of hardware and software results. First, we remark that each probe, investigated by PROLEAD is glitch-extended and therefore expands to a probing set whose size is bounded by the underlying circuit. Contrary, each probe in

the software scenario expands to a fixed-size probing set making the number of required simulations more predictable. Further, if PROLEAD\_SW evaluates an implementation without considering horizontal or vertical probes, each probe expands to at most two standard probes, i.e., when the standard probe is transition-extended. This leads to confident results based on a few thousand simulations which are usually not feasible for hardware. On the other side, considering all micro-architectural effects, especially by including horizontal or vertical probes, requires significantly more simulations to achieve a confident result.

## 5 Case Studies

To show the importance of PROLEAD\_SW for the evaluation of masked software and to be able to make practical statements concerning its performance, we examined real-world case studies based on publicly available implementations. In summary, PROLEAD\_SW detects security flaws in almost all investigated implementations under our proposed security models. Based on the reports of PROLEAD\_SW, we can precisely isolate the leakage source and give practical hints for avoidance. A summary of the considered implementations and details of the detected leakages including the required resources and the performance of PROLEAD\_SW are given in Table 1.

### 5.1 Setup

We evaluated all implementations based on a fixed-vs-random setting. Hence, we considered two groups, one with a fixed vector as input, and the other one a random input. For all AES implementations we used the fixed plaintext `0xda39a3ee5e6b4b0d3255bfe95601890`, and the key `0x2b7e151628aed2a6abf7158809cf4f3c`. Our general procedure is as follows. First, we evaluate all implementations by considering normal probes only. If PROLEAD\_SW detects no leakage, we continue the evaluation considering transitions as well as probes on the RAM. Again, if no leakage is detected, we rerun the evaluation and, in addition, consider horizontal and vertical probes. Finally, if PROLEAD\_SW does not report any leaking probes, we consider the implementation as secure. As we want to be comparable to the similar experimental analysis reported in [BWG<sup>+</sup>22], we restrict the number of simulations to 10 000. If we encounter implementations that show leakage in the standard ISW model we omit the evaluation of further effects and cease an evaluation if we identify significant leakage that increases after every simulation.

We ran PROLEAD\_SW in a Linux subsystem, containing a Ubuntu 20.04 LTS 64-bit distribution, on a Windows server with an AMD EPYC 7352 CPU. While the CPU contains 48 hyper-threading cores operating at 2.3 GHz, we limited the number of parallel threads to 30. Moreover, the server has 500 GB of RAM, which is more than enough to keep all probing sets, including contingency tables, in the RAM. As all experiments are possible with a smaller amount of RAM, we report the RAM usage for each experiment separately. We compiled the source files with the 9-2019-q4-major `arm-none-eabi-gcc` version.

### 5.2 Provable Secure Software Masking in the Real-World

We start with investigating a collection of four AES implementations from different authors experimentally evaluated in [BWG<sup>+</sup>22]. The underlying masking schemes differ across the implementations. For their experimental analysis, the authors of [BWG<sup>+</sup>22] imitated the compiler optimizations from the given makefiles of the original implementations/repositories. All of their software was compiled with `-O3` except the Inner Product Masking (IPM) scheme, which was compiled with `-O1`. We follow the same principle for compilation and evaluation to allow a meaningful comparison of the results. In short, the authors

detected first-order leakage for all implementations based on a Test Vector Leakage Assessment (TVLA) with 10 000 traces on an STM32F415 micro-controller with Cortex-M4.

### 5.2.1 Provably Secure Higher-Order Masking of AES

Rivain and Prouff [RP10] adapted the hardware-oriented masking scheme by Ishai et al. [ISW03] for software and extended it to higher security orders. More specifically, they generalized the protected multiplication from Ishai et al. [ISW03] over  $\mathbb{F}_2$  to  $\mathbb{F}_2^n$ . Further, they implemented the non-linear part of the AES Sbox computation, i.e.,  $x^{254}$  in  $\mathbb{F}_2^8$ , by replacing all unprotected multiplications of the multiplication chain with their secure version. A reference implementation for arbitrary security orders is publicly available on GitHub<sup>8</sup>. We remark that the given masking scheme is provably  $d$ -probing secure.

**Results.** Without considering physical effects and memory probes, PROLEAD\_SW confirms its  $d$ -probing security using on 10 000 simulations. However, if PROLEAD\_SW covers leakages due to transitions, i.e., evaluating the MOE-security, a few hundred simulations are enough to detect significant first-order leakage. We analyzed the source of leakage in depth using PROLEAD\_SW’s provided information and visualized it with the provided disassembled code snippet.

```

<multshare.constprop.0>:
[...]
(496) eors   r3, r1 //calculate result, store one share in r3
[...]

<cipher>:
[...]
    bl      0x80f8 <multshare.constprop.0>
(500) ldrb.w r2, [sp, #181]
(501) ldrb.w r3, [sp, #180] //Write second share in r3
[...]

```

The multshare function performs a shared multiplication resulting in two output shares. Instruction 496 writes one of the output shares into register  $r_3$  before terminating the function. After termination, instruction 501 stores a value from memory into register  $r_3$  overwriting the shared output of multshare. However, the memory state depends on the second share leading to transitional leakage during the overwrite on  $r_3$ . We point out that transitional leakage does not violate the author’s claim of  $d$ -probing security. However, transitional leakage due to a register overwrite is not bounded to a specific CPU but makes the implementation insecure in practice and for arbitrary CPUs. Hence, a practically secure implementation must avoid transitional leakages due to memory overwrites.

### 5.2.2 Higher order Masking of Look-up Tables

Contrary to [RP10], the construction of Coron does not compute any multiplication in  $\mathbb{F}_2^8$  [Cor14] but generalizes the randomized table countermeasure [CJRR99] to arbitrary orders. The idea of the author in the first-order case is to start randomizing the Sbox table  $S(x)$  in memory by computing  $T(x') = S(x \oplus r) \oplus s$ , with input mask  $r$  and output mask  $s$ . A table look-up with  $x' = x \oplus r$  being the masked input is then the memory access of  $T(x')$ . This process of recomputing the randomized table with inputs blinded by  $r$  followed by a masked table look-up is repeated for every access to  $T$ . The underlying implementation can be found on GitHub<sup>9</sup>.

<sup>8</sup><https://github.com/coron/htable>

<sup>9</sup><https://github.com/coron/htable>

**Results.** PROLEAD\_SW confirms the first-order probing security of the implementation using 10 000 simulations when considering only normal probes. Nevertheless, PROLEAD\_SW detects a violation under the MOE-security running a few hundred simulations. Again, using extra information provided by PROLEAD\_SW enabled us to track security violations, such as the transition of two shares in the following code snippet. The leakage is caused by far-reaching compiler optimizations. In particular, the compiler merges ShiftRows and MixColumns operations. Consequently, MixColumns accesses the bytes as if they were already shifted and writes the result to the correctly shifted position. In this context, instruction 70152 loads one share from memory into register  $r_0$  on which linear operations are performed.

```
<cipher>:
[...]
(70152) ldr.w    r0, [r7, #128] // load one share
[...] // code does not change content of r0
(70163) ldrb.w  r0, [fp, #15] // load second share
[...]
```

Afterwards, the function linearly processes the second share after writing it into  $r_0$ . Therefore, the transition on  $r_0$  leaks information about the unshared state. For this example, PROLEAD\_SW detects such transitions across multiple instructions with different distances in time. Again, the implementation becomes insecure on arbitrary CPUs due to the memory overwrite effect.

### 5.2.3 Side-Channel Masking with Pseudo-Random Generator

The third masking scheme, which is also available on GitHub<sup>10</sup>, adapts the Rivain-Prouff countermeasure in addition to a Final Locality Refresh (FLR) in each gadget [CGZ20]. The randomness locality  $l$  depicts the amount of random bits required to calculate each value in the circuit with only the original input and at most  $l$  bits of randomness [IKL<sup>+</sup>13]. The goal of improving the locality is to require less random bits in total. To create the required randomness different independent PRNGs are instantiated.

```
<multishare_flr_mprgmat>:
[...]
(537) ldrb    r3, [r5, #0] //contains first share
(538) ldrb    r2, [r5, #1] //contains second share
(539) strb    r0, [r5, #0]
(540) eors    r3, r2      //recombination of shares
[...]
```

We confirm the observation in [BWG<sup>+</sup>22] that the underlying masking scheme on the compiled binary provides no security in the first round. In fact, we could verify that the compiler causes undesired recombination of the shares by XOR-ing the corresponding registers together, which is already insecure in the standard ISW-model. When evaluating by only normal probes, PROLEAD\_SW's provided information helped us to identify the snippet shown above as a hazardous example. After loading the two shares in  $r_3$  and  $r_2$ , instruction 540 reveals the unshared value. Strictly speaking, this is not a micro-architectural leakage but a security flaw in the implementation. Thus, this is independent of the employed platform.

### 5.2.4 Detecting Faults in Inner Product Masking (IPM) Scheme

Cheng et al. [CCG<sup>+</sup>21] combined IPM and fault detection techniques to create a masked AES implementation<sup>11</sup>, which also detects faults. As PROLEAD\_SW cannot detect faults, we solely concentrate on the IPM scheme. To represent variable  $x$  in a masked form, two random vectors  $L = (l_1, \dots, l_n)$  and  $R = (r_1, \dots, r_n)$  are constructed, such that  $x$  is the

<sup>10</sup><https://github.com/coron/htable>

<sup>11</sup><https://github.com/Qomo-CHENG/IPM-FD>

inner product of the vectors  $L$  and  $R$  [BFGV12]. Cheng et al. further adapted inner product operations such as addition, multiplication, and refresh to their needs to handle the combination of masking and fault detection.

**Results.** PROLEAD\_SW detects first-order leakage within a set of 10 000 simulations in the ISW-model, i.e., no transitional, horizontal, or vertical extension. However, the observed leakage, i.e., the  $g$ -statistic value, grows slower compared to previously evaluated implementations. All investigated leaking instructions are located within the GF256\_Mult function, computing multiplications in  $\mathbb{F}_2^8$ , but from different function calls. We give the source code of GF256\_Mult below.

```
uint8_t GF256_Mult(uint8_t a, uint8_t b) {
    int x = a, y = b, m, res = 0;
    for (unsigned char i = 0; i < 8; i++) {
        m = -(y & 1); //m is either 0xffff or 0x0000
        res ^= (x & m);
        y >>= 1;
        x <<= 1;
        m = -((x >> 8) & 1); //MSB
        x ^= (m & 0x1b);
    }
    return (uint8_t)res;
}
```

Concretely, the XOR operation (highlighted in red) causes leakage due to a potentially non-uniform sharing. We note that the leakage only occurs during particular function calls, e.g. if the input sharing is not uniform. However, this violates the practical security independent of the underlying platform.

### 5.3 Hardened Library for AES-128 Encryption/Decryption on ARM Cortex-M4 Architecture

The French government agency referred to as ANSSI<sup>12</sup> made use of affine masking [FMPR10] to create a first-order protected AES implementation available via GitHub<sup>13</sup>, also known as ASCAD\_v2. Its (input and key) state  $X \in (\mathbb{F}_2^8)^{16}$  is of the form  $r \times x_i \oplus m_i$  while  $r \in \mathbb{F}_2^8$  denotes a non-zero random value as multiplicative mask,  $x_i \in \mathbb{F}_2^8$  one byte of the state, and  $m_i \in \mathbb{F}_2^8$  one byte of a random 128-bit vector. The finite-field multiplication in  $\mathbb{F}_2^8$  using Rijndael polynomial is indicated by  $\times$ . Further, the implementation combines masking with random shuffling of the operations order. An encryption procedure is divided into three steps. First, the pre-processing step performs precomputations of the tables and the masked key schedule. Second, actual AES round computations are carried out. Finally, the post-processing step unmaskes the state to generate a valid ciphertext. The authors experimentally verified the SCA resistance of their implementation on a ChipWhisperer board based on an STM32F303RCT7 chip with a Cortex-M4 core by performing CPA attacks on 50 000 traces.

**Results.** This example demonstrates the ability of PROLEAD\_SW to handle pre-compiled binaries, i.e. providing a binary as input. The authors provided a makefile generating a ready-to-use firmware binary for the STM32F4 discovery board. We used the same makefile to generate the binary outside of PROLEAD\_SW, i.e. before invocation. We only commented out hardware-specific operations, e.g., flushing the board, initializing LEDs etc., and added the generation of the map and disassembled files. We provided PROLEAD\_SW with the path to our binary, map file and disassembled file and started the evaluation process. The result of our first-order assessment did not show any leakage for 10 000 simulations under

<sup>12</sup><https://www.ssi.gouv.fr/>

<sup>13</sup><https://github.com/ANSSI-FR/SecAESSTM32>

any of our proposed leakage models. Therefore, we confirm the first-order security of the underlying implementation.

## 5.4 Bitslicing Arithmetic/Boolean Masking Conversions

Efficient conversions between arithmetic and Boolean masking are of great importance for the performance of lattice-based Key Encapsulation Mechanisms (KEMs). Such post-quantum schemes combine bit-level logical with arithmetic operations modulo  $p$ . Both Boolean and arithmetic masking achieve performance improvements in their respective field. Bronchain and Cassiers [BC22] introduced efficient masked bitsliced Arithmetic-to-Boolean (A2B) and Boolean-to-Arithmetic (B2A) conversion gadgets for arbitrary orders. Additionally, the authors introduced a gadget computing addition over Boolean masked variables without any conversions. All gadgets are secure under the Probe-Isolating Non-Interference (PINI) notion [CS20] and are hardened to cope with micro-architectural effects, such as transitions or overwrites in memory paths.

We used the dedicated test cases, provided by the authors on GitHub<sup>14</sup> for the evaluation of all three gadgets used in their K3 and S3 implementation of Kyber [BDK<sup>+</sup>18], respectively Saber [DKRV18]. For the sake of consistency with other parts of the paper, we evaluated the first-order security of all gadgets, i.e., with two shares, and using only 10 000 simulations. Moreover, we set the modulus to  $p = 3329$ . We imitated the compiler flags for the test cases except for the floating point option, which is currently not supported by PROLEAD\_SW.

### 5.4.1 Secure Addition

The secure addition function `SecAdd` implements a key component of the following conversion gadgets.

**Results.** By PROLEAD\_SW we confirm its 1-probing security, as no leakage was found within a set of 10 000 simulations using the standard ISW-model. However, PROLEAD\_SW detects significant transitional leakages after processing a few hundred simulations. We identified the most significant bit of the Program Status Register (PSR) as the source of leakage. Therefore, we searched for instructions updating the PSR in the manually written assembly code and identified two occurrences.

```
<loop_cross_ext>:
[... ]
eors r2, r2, r0    // r2 = bj ^ r, updates PSR
[... ]
eors r2, r2, r0    // r2 = bj ^ r, updates PSR
[... ]
```

Here, both instructions overwrite the most significant bit of the PSR with the most significant bit of the instructions' result. Consequently, the PSR leads to transitional leakage if the given instructions operate on both shares separately, thus breaking the MOE-security. To avoid transitional leakage, we remove the status flag specifier, i.e., we replace all `eors` with `eor`. By this the correct functionality of the gadget is still maintained, and we did not detect any transitional leakages by up to 10 000 simulations. In fact, under any of our proposed models, PROLEAD\_SW still detects no leakages. For the following evaluations, we removed the status update flag from all such instructions.

### 5.4.2 Secure Arithmetic-to-Boolean Masking Conversion Modulo $p$

The A2B gadget follows a recursive divide-and-conquer approach. The gadget converts two halves of the shares to their Boolean form and adds them by applying the secure addition

<sup>14</sup>[https://github.com/uclcrypto/pqm4\\_masked](https://github.com/uclcrypto/pqm4_masked)

modulo  $p$ .

**Results.** As before, PROLEAD\_SW detects no leakage even considering MRE-security and BIL-security. However, contrary to the experimental results shown in [BC22], PROLEAD\_SW detects first-order leakage after a few hundred simulations if we evaluate the the NLE-security. We used PROLEAD\_SW’s output to trace the corresponding vertical probe back to its assembly instruction. The first standard probe is inside the `secadd_constant` function, which gets called internally. Inside this function a local array is declared, which during the execution contains the copy of the bit-wise inversion of one input share. On the assembly level, this translates to an instruction storing the contents on register  $r_{14}$ . The content of  $r_{14}$ , i.e., the bit-wise inversion of the share, does not get cleared until we reach the second probe of our probing set. This probe contains the second input share by loading the value into register  $r_4$  during the `copy_sharing_loop` function call, subsequent to `secadd_constant` function. When we now place a vertical probe on  $r_4$ , which also receive information about  $r_{14}$ , we indeed created a probing set which combines both shares.

```

<secadd_constant>:
[...]
(1750) str r14, [r8]
[...]
<copy_sharing_loop>:
[...]
(2696) ldr r4, [r3, #0]
[...]

```

The reason why the authors did not experimentally see such a leakage can be threefold. PROLEAD\_SW does not assume a specific target architecture but evaluates a generic model. As our model is designed to capture all possible occurrences of specific physical defaults (in this case, neighbor leakage effects), it tends to be over-conservative. A particular neighbor leakage effect does not necessarily occur on every device. Here, the concrete occurrence of leakages depends on the implementation of the multiplexer-tree between registers and the ALU. Nevertheless, PROLEAD\_SW makes it possible to evaluate physical security for *arbitrary* designs and *independent* of the targeted CPU. In other words, the leakage detected by PROLEAD\_SW might be observed in practice if the experiments are conducted on a different CPU than that of [BC22]. Further, the underlying evaluation schemes differ slightly, as we evaluate the implementation with a probing-based fixed-vs-random test, while the authors have evaluated their implementation with a fixed-vs-fixed t-test. Lastly, we note that the compiler flags do not perfectly match the ones used by Bronchain and Cassiers as M-Ulator lacks a Floating Point Unit (FPU) and, thus, the compiled binaries might be different.

### 5.4.3 Secure Boolean-to-Arithmetic Masking Conversion Modulo $p$

Contrary to all previous case studies, the B2A gadget is not constant-time as the underlying algorithm probabilistically samples randomness from  $\mathbb{Z}_p$ .

**Results.** PROLEAD\_SW can also handle implementations that are not constant-time in a reasonable amount of time. If we test the MOE-security, MRE-security and BIL-security, PROLEAD\_SW detects no leakages after simulating 10 000 runs. However, when we evaluating the NLE-security, we detect first-order leakages similar to the above explained test on the A2B conversion gadget. This stems from multiple internal calls to the arithmetic to Boolean conversion function evaluated in the previous section.



## 5.5 Threshold Implementations in Software: Micro-Architectural Leakages in Algorithms

Gaspoz and Dhooge [GD22] designed first-order secure software threshold implementations by achieving register non-completeness and uniformity based on the properties described in Lemma 1 and Lemma 2. They provided a first-order masked PRESENT implementation, two variants of the Keccak-f permutation, and the masked representation of 4-bit quadratic bijective classes. Security of all implementations were experimentally verified on a CW308T-STM32F target board with a STM32F415RG Cortex-M4 micro-controller. They collected one million measurements and performed fixed-vs-random first-order t-tests. All implementations stayed within the threshold boundaries. Their implementations are publicly available on GitHub<sup>15</sup>. We evaluated all schemes with 10 000 simulations under our generic leakage models. While we could verify the non-completeness and uniformity properties of the quadratic classes and Keccak-f implementations, we identified leakage from vertical probes in the PRESENT implementation, which violates the NLE-security.

### 5.5.1 PRESENT

The masked cipher takes two shares of the 64-bit plaintext as inputs. In order to ensure register non-completeness, the authors have shifted one share of the sensitive values one bit to the right. Then, the permutation layer of PRESENT is performed share-wise. As the non-linear layer contributes to the highest overhead in terms of runtime, the authors performed the Sbox computations in parallel by bitslicing the shares and calculating the output based on the algebraic normal form of the component functions.

**Results.** With PROLEAD\_SW, we detected subtle leakages across multiple instructions that violate the register non-completeness property with only a few hundred simulations. More specifically, we are able to find vertical probes that combine bits of both shares during the bitslice generation for the Sbox inputs. After the `orr` instruction in line 121 shown below,  $r_2$  contains the least significant bits of all state nibbles of the first share. As follows, we visualize the content of  $r_2$  where each position of the register contains the symbolic representation of the share and its corresponding bit index.

```
<sbox_player>:
[... ]
(121) orr r2, r2, r11
[... ]
(469) lsr r11, r10, #9
[... ]
```

31 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
unused	$s_{32}^0$	$s_{36}^0$	$s_{40}^0$	$s_{44}^0$	$s_{48}^0$	$s_{52}^0$	$s_{56}^0$	$s_{60}^0$	$s_0^0$	$s_4^0$	$s_8^0$	$s_{12}^0$	$s_{16}^0$	$s_{20}^0$	$s_{24}^0$	$s_{28}^0$

The content of  $r_2$  does not change until we reach the `lsr` instruction in line 469. Here,  $r_{10}$  contains the higher 32 bits of the second share. Because the second share is rotated by one bit, we can symbolically represent the contents of  $r_{10}$  as  $s_{32}^1$  at the most significant bit position, followed by  $s_{63}^1$ ,  $s_{62}^1$ ,  $s_{61}^1$  and so forth until we end up with  $s_{33}^1$  at the least significant bit position. If we shift  $r_{10}$  now by nine positions to the right and write the result into  $r_{11}$  (as done by the instruction in line 469), we end up with the state given below.

31	30	29 ... 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$s_{41}^1$	$s_{40}^1$	...	$s_{55}^1$	$s_{54}^1$	$s_{53}^1$	$s_{52}^1$	$s_{51}^1$	$s_{50}^1$	$s_{49}^1$	$s_{48}^1$	$s_{47}^1$	$s_{46}^1$	$s_{45}^1$	$s_{44}^1$	$s_{43}^1$	$s_{42}^1$

<sup>15</sup><https://github.com/KULeuven-COSIC/Software-masked-Keccak-and-PRESENT>

We can see that at index 10 of registers  $r_2$  and  $r_{11}$  both shares of bit 52 are vertically at the same position, thus violating the register non-completeness property. Again, the question whether the leakage is observable in practice depends on the way the ALU is designed and constructed.

**Table 1:** Evaluation results. Effects represent the currently evaluated security model and are abbreviated as in Section 4.1.1.

Design	Effects							#Instr [in K]	#Sets [in Mio.]	Performance	
	ISW	MOE	MRE	BIL	NLE	PRO	PFE			[RAM]	[time]
AES, RP [RP10]	✓	✗	✗	✗	✗	✗	✗	11.5	0.74	5.5 GB	14 s
AES, Htable [Cor14]	✓	✗	✗	✗	✗	✗	✗	71	4.16	9.8 GB	1 m
AES, IPM [CCG+21]	✗	✗	✗	✗	✗	✗	✗	113.3	8.12	13 GB	35 m
AES, FLR [CGZ20]	✗	✗	✗	✗	✗	✗	✗	25.2	1.56	5.8 GB	22 s
ANSSI AES*[FMPR10]	✓	✓	✓	✓	✓	✓	✓	2.8	0.35	21 GB	17 m
SecAdd [BC22]	✓	✓	✓	✓	✓	✓	✓	4	3.80	18.3 GB	27 m
SecB2AModp [BC22]	✓	✓	✓	✓	✗	✗	✗	variable <sup>†</sup>	89.68	123 GB	21 h
SecA2BModp [BC22]	✓	✓	✓	✓	✗	✗	✗	9.8	9.64	45 GB	10 m
PRESENT [GD22]	✓	✓	✓	✗	✗	✗	✗	2.7	1.40	14 GB	4 m
Keccak-f1600 [GD22]	✓	✓	✓	✗	✓	✓	✓	3.8	0.31	95 GB	29 m
Keccak-f800 [GD22]	✓	✓	✓	✗	✓	✓	✓	2	0.16	45 GB	14 m
Quadratic Class [GD22]	✓	✓	✓	✗	✓	✓	✗	0.3	0.03	8 GB	3 m

\* specific GitHub implementation<sup>13</sup>

<sup>†</sup> on average approximately 50 000 instructions

## 6 Limitations

While PROLEAD\_SW enables the evaluation of large software designs, which are out of the scope of the known formal verification tools, there is still plenty of work to be done. Below, we discuss all features missing in M-Ulator and PROLEAD\_SW. Integrating these features can help to further improve assessment of masked software implementations.

### 6.1 M-Ulator Limitations

Currently, M-Ulator supports only the instructions of ARMv6-M, ARMv7-M, and ARMv7E-M ISAs. Other architectures, e.g., ARMv8-M, are not supported yet. Further, the utilization of floating point registers as temporary storage elements may reduce the runtime and enable higher flexibility, especially for post-quantum schemes [ACC<sup>+</sup>21, CHK<sup>+</sup>21], but cannot be emulated as the FPU is not supported by M-Ulator. Almost all CPUs that implement the currently supported ISAs do not come with built-in caches. Contrary, the Cortex-M7 implementing the ARMv7E-M ISA utilizes a 64 kB instruction and data cache which is not supported by M-Ulator.

### 6.2 PROLEAD\_SW Limitations

The evaluation procedure is generic in the sense that the CPU-netlist is not required. In other words, PROLEAD\_SW does not ask for any detailed information about the exact physical behavior of the CPU. Consequently, if all micro-architectural effects (transitions, vertical, horizontal probes, etc.) are enabled, its evaluation tends to be conservative as it analyzes implementations in a CPU-independent leakage model, i.e., a CPU whose detailed internal architecture is not known. Therefore, the leakages detected by PROLEAD\_SW may not be observable in practice for every realization/implementation of the underlying CPU.

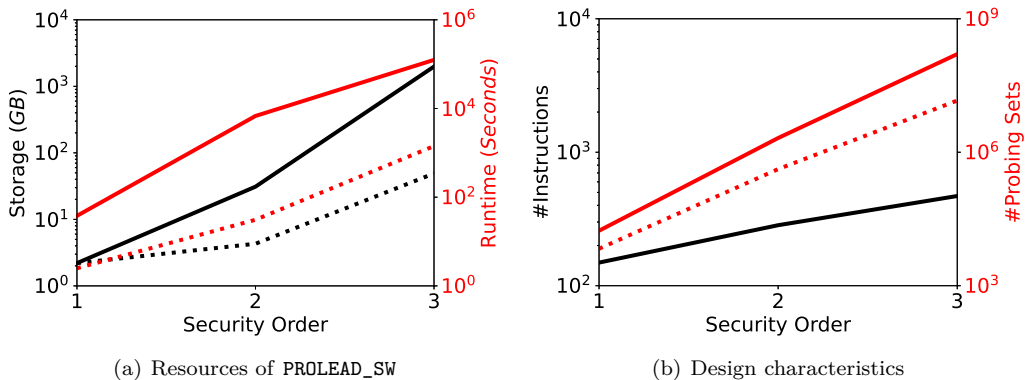
However, if PROLEAD\_SW reports the security of an implementation when covering all micro-architectural effects, the implementation would very likely stay secure in practice.

To the best of our knowledge, no other generic formal verification tool considers more micro-architectural effects than PROLEAD\_SW, but it even still misses some micro-architectural sources of leakage. These include but not limited to leakages due to branch prediction, speculative execution, and caches. Covering such missing micro-architectural effects, i.e., defining their respective models and adding them to our generic leakage model, would be a beneficial improvement to PROLEAD\_SW. Similar to [PV17], we believe that a single probing set encompassing standard probes on all bits of every register at the same time would be able to cover more, even unnoticed, micro-architectural effects. However, such a model limits the designer to store only an incomplete set of shares in the entire registers and requires an infeasible amount of simulations and/or computing resources to get an insight about the security/vulnerability of the implementation under test. In fact, evaluating the implementations from [GD22] under this additional restriction exhibits leakage as all of the shares are simultaneously present in the register banks.

In the context of performance, PROLEAD\_SW can analyze the security of first-order masked implementations within minutes to hours as shown in Section 5. The analysis of higher-order schemes forthright follows the approach in Section 4.6 but is computationally more intense, especially in multivariate scenarios.

### 6.3 Benchmarks

Finally, we show the capabilities of PROLEAD\_SW by evaluating a masked implementation computing a bitsliced Boolean AND operation receiving two 32-bit operands and generating the corresponding 32-bit output. The implementation is publicly available via GitHub<sup>16</sup>, and can be compiled for arbitrary security orders, i.e. with different numbers of shares, and is hardened against all micro-architectural leakage sources considered by PROLEAD\_SW. Consequently, we detect no univariate leakage for the first three security orders using 1000 simulations. The benchmarking results, visualized in Figure 7, are twofold. First, we conducted the evaluations without considering any micro-architectural effects (indicated by dotted lines). Then, we enabled the detection of violations in all of our covered micro-architectural leakage models and reran the evaluations (indicated by solid lines).



**Figure 7:** Benchmark results of PROLEAD\_SW for different security orders using 1000 simulations. Curves with dotted lines indicate the results when no micro-architectural effects are considered, and curves with solid lines when all micro-architectural effects are covered.

The runtime and the required RAM of PROLEAD\_SW grow exponentially with the security

<sup>16</sup>[https://github.com/uclcrypto/pqm4\\_masked](https://github.com/uclcrypto/pqm4_masked)

order which is due to the exponentially increasing number of probing sets and size of the underlying contingency tables. Therefore, we guess that the evaluation of larger masked designs with PROLEAD\_SW is limited to the first two orders when multivariate leakages are expected to be covered. However, similar to PROLEAD for hardware, the evaluation can be restricted to particular parts of the code, particular probing sets, and particular micro-architectural effects. Hence, even if a full evaluation with all micro-architectural effects on the entire software is not feasible, PROLEAD\_SW may help software engineers to find potential security flaws.

## 7 Conclusions

In this work, we presented PROLEAD\_SW, an extension of the hardware-based leakage evaluator PROLEAD to verify the probing security of masked software implementations in our newly presented CPU-independent leakage model. More concretely, PROLEAD\_SW targets software implementations for Cortex-M processors at the binary level. Its highly-accelerated simulation-based approach – based on the open-source ARM emulator M-Ulator – allows the assessment of large implementations in a reasonable amount of time, which is infeasible when using state-of-the-art formal verification tools. To support a broad range of applications, we extended M-Ulator by additionally supporting the ARMv7E-M ISA excluding floating point operations. Contrary to other leakage simulators, we are not bounded to a hypothetical power consumption model, and can handle arbitrary logic in implementations. More specifically, PROLEAD\_SW can handle recursions, branches, conditions, and non constant-time codes. PROLEAD\_SW can detect a wide range of micro-architectural effects leading to some form of glitches and transitions while being in line with the robust probing security model. We have shown the efficiency, accuracy, as well as importance of PROLEAD\_SW through multiple case studies. By means of PROLEAD\_SW, we identified leakage due to several micro-architectural effects that are covered in our security models during the execution of binaries leading to security degradation in several publicly available masked software implementations. We highlight that we confirmed the findings of PROLEAD\_SW by carefully examining its reports, i.e., the identified source of leakages.

## Acknowledgments

The work described in this paper has been supported in part by the German Research Foundation (DFG) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972, and by the Federal Ministry of Education and Research of Germany through the Project DevToSCA (16KIS1603).

## References

- [ACC<sup>+</sup>21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):217–238, 2021.
- [BBC<sup>+</sup>19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults. In *ESORICS 2019*, volume 11735 of *LNCS*, pages 300–318, 2019.

- [BBD<sup>+</sup>15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified Proofs of Higher-Order Masking. In *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 457–485. Springer, 2015.
- [BBD<sup>+</sup>16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In *ACM CCS 2016*, pages 116–129. ACM, 2016.
- [BBYS22] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. SoK: Design Tools for Side-Channel-Aware Implementations. In *ASIA CCS 2022*, pages 756–770. ACM, 2022.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.
- [BDK<sup>+</sup>18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *EuroS&P, 2018*, pages 353–367. IEEE, 2018.
- [BDM<sup>+</sup>20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In *EUROCRYPT 2020*, volume 12107 of *LNCS*, pages 311–341, 2020.
- [BFGV12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. Theory and Practice of a Leakage Resilient Masking Scheme. In *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 758–775. Springer, 2012.
- [BGI<sup>+</sup>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 *EUROCRYPT 2018*, volume 10821 of *LNCS*, pages 321–353. Springer, 2018.
- [BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight Private Circuits: Achieving Probing Security with the Least Refreshing. In *ASIACRYPT 2018*, volume 11273 of *LNCS*, pages 343–372. Springer, 2018.
- [BWG<sup>+</sup>22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable Secure Software Masking in the Real-World. In *COSADE 2022*, volume 13211 of *LNCS*, pages 215–235. Springer, 2022.
- [CCG<sup>+</sup>21] Wei Cheng, Claude Carlet, Kouassi Goli, Jean-Luc Danger, and Sylvain Guilley. Detecting faults in inner product masking scheme. *J. Cryptogr. Eng.*, 11(2):119–133, 2021.
- [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 *COSADE 2018*, volume 10815 of *LNCS*, pages 82–98. Springer, 2018.
- [CGZ20] Jean-Sébastien Coron, Aurélien Greuet, and Rina Zeitoun. Side-Channel Masking with Pseudo-Random Generator. In *EUROCRYPT 2020*, volume 12107 of *LNCS*, pages 342–375. Springer, 2020.

- [CHK<sup>+</sup>21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.
- [CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
- [Cor14] Jean-Sébastien Coron. Higher Order Masking of Look-Up Tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, 2014.
- [CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.
- [DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying Leakage Models: From Probing Attacks to Noisy Leakage. In *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, 2014.
- [dGHM22] Arnaud de Grandmaison, Karine Heydemann, and Quentin L. Meunier. ARMISTICE: microarchitectural leakage modeling for masked software formal verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(11):3733–3744, 2022.
- [DKRV18] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM. In *AFRICACRYPT 2018*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018.
- [FGP<sup>+</sup>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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.
- [FMPR10] Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. Affine Masking against Higher-Order Side Channel Analysis. In *SAC 2010*, volume 6544 of *LNCS*, pages 262–280. Springer, 2010.
- [GD22] John Gaspoz and Siemen Dhooghe. Threshold Implementations in Software: Micro-architectural Leakages in Algorithms. *IACR*, page 1546, 2022.
- [GHP<sup>+</sup>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 *USENIX Security 2021*, pages 1469–1468, 2021.
- [GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *CHES 2001*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001.
- [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.

- [HSP21] Max Hoffmann, Falk Schellenberg, and Christof Paar. ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries. *IEEE Trans. Inf. Forensics Secur.*, 16:1058–1073, 2021.
- [IKL<sup>+</sup>13] Yuval Ishai, Eyal Kushilevitz, Xin Li, Rafail Ostrovsky, Manoj Prabhakaran, Amit Sahai, and David Zuckerman. Robust Pseudorandom Generators. In *ICALP 2013*, volume 7965 of *LNCS*, pages 576–588. Springer, 2013.
- [ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.
- [KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
- [Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO 1996*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
- [MM22] Nicolai Müller and Amir Moradi. PROLEAD A Probing-Based Hardware Leakage Detection Tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 2022.
- [MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software. *IACR Cryptol. ePrint Arch.*, page 1297, 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 *USENIX Security 2017*, pages 199–216, 2017.
- [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.
- [PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In *COSADE 2017*, volume 10348 of *LNCS*, pages 282–297. Springer, 2017.
- [RP10] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In *CHES 2010*, volume 6225, pages 413–427. Springer, 2010.
- [Sha79] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.
- [SSB<sup>+</sup>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 *NDSS 2021*, 2021.
- [Yiu16] Joseph Yiu. ARM Cortex-M for Beginners, 2016. [https://community.arm.com/cfs-file/\\_\\_key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-96/White-Paper-\\_2D00\\_-Cortex\\_2D00\\_M-for-Beginners-\\_2D00\\_-2016-\\_2800\\_final-v3\\_2900\\_.pdf](https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-96/White-Paper-_2D00_-Cortex_2D00_M-for-Beginners-_2D00_-2016-_2800_final-v3_2900_.pdf), accessed on Jan. 2023.