PROLEAD_SW

Probing-Based Software Leakage Detection for ARM Binaries

Jannik Zeitschner*, Nicolai Müller* and Amir Moradi

Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
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 which 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 satisfying 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 to make general and device-independent statements about its security.

In this work, we adapt PROLEAD for the evaluation of masked software, which has recently been presented at CHES 2022 and originally developed as a simulation-based tool to evaluate masked hardware designs. We enable to transfer 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 the well-established robust probing security model. In short, together with an abstraction model for the micro-architecture, the robust probing model allows us to efficiently detect micro-architectural leakages while being independent of a concrete CPU design. As a concrete result, using PROLEAD_SW we evaluated the security of several publicly available masked software implementations and revealed multiple vulnerabilities.

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

1 Introduction

Nowadays, researches 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 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]. 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].

*These authors contributed equally to this work.

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+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+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].

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 a 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 complete masked AES core, without resorting to inaccurate heuristics. 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 adapt the working principle of PROLEAD to evaluate masked software implementations independent of an actual CPU design. The resulting framework PROLEAD_SW is publicly available via GitHub[1]and extends the original PROLEAD with the ability to evaluate the robust d-probing security 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] https://github.com/ChairImpSec/PROLEAD

1.2.1 Formal Verification

The purpose of `maskVerif` [BBC+19] is to verify higher-order masked software for probing-security, NI [BBD+15], and SNI [BBD+16] notions under the ISW 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+20] aims to generate provable secure masked and bitsliced C code for arbitrary orders under the d-probing model, respectively register probing model [BDM+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+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+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 practical statements about the implementation's security. Nevertheless, `COCO` cannot evaluate software with a non-constant control flow as well as arithmetic masking.

1.2.2 Leakage Simulation

The goal of `ASCOLD` [PV17] is to detect the violation of Independent Leakage Assumption (ILA) by simulated information flow tracking. The tool considers 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 are able to 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 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+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 comes from the executed instructions or the underlying hardware. Given the assembly-level source code and an 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.

Usability. RootCanal and COCO suffer from the same downside as both require the hardware netlist (which is not publicly available in several cases including Advanced Risc Machines (ARM)) limiting their application mostly to pre-tapeout verifications. We remark that PROLEAD_SW requires only the binary to analyze, which expands its applicability. Further, ASCOLD, MAPS, and ELMO implement only a subset of their underlying Instruction Set Architecture (ISA). This obviously limits the high-level functionality of the implementations which can be examined. As an (incomplete) example, ASCOLD supports no loops, MAPS supports no conditional branches, and ELMO implements only a subset of 21 instructions. In contrast, PROLEAD_SW supports the complete ARMv7E-M ISA[2]. More concretely, none of the aforementioned simulators (ASCOLD, MAPS, ELMO) is applicable to the examples we give in Section 5 without severe modifications either to the simulator or to the examples to be in line with the respective simulator boundaries.

Completeness. RootCanal and COCO can identify processor-specific leakages most accurately due to their white-box approach. They do not have to define micro-architectural effects, but directly track leakages back to the hardware source. While ELMO is a grey-box tool, it does not pre-define the detectable micro-architectural effects but rather handles micro-architectural behavior that are observable during the pre-processing step. Besides being non-trivial to generate the model and tailored to one specific micro-controller, ELMO depends on the accuracy of the measurements, i.e., order of instructions, amount of traces, and applied modeling technique. In the case of TightPROVE+, no micro-architectural effects are covered, while maskVerif can only cover transitions, leaving other known effects undetected. While MAPS and ASCOLD deal with some security reduction effects, their evaluation is limited to first-order designs. Contrary, PROLEAD_SW incorporates more micro-architectural leakage effects than TightPROVE+, maskVerif, MAPS, and ASCOLD while being independent of the applied CPU and applicable to higher order.

Performance. As formal verification is exhaustive, meaning that it checks all possible input vectors, the evaluation of larger implementations becomes infeasible. As an example, we consider the software case studies of maskVerif available online[3]. 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 tools do not verify the actual instructions executed on a CPU but an abstract representation. PROLEAD_SW removes any uncertainty regarding translations to intermediate representations or compiler optimizations. Nevertheless, PROLEAD_SW and other leakage simulators (except ASCOLD) are not exhaustive. Hence, they cannot prove the

[2] Floating point instructions are excluded, but it should not limit any cryptographic implementations.
[3] https://cryptoexperts.com/maskverif/

security of a given implementation and their accuracy depends on the number of simulations, as for ELMO and MAPS. However, PROLEAD_SW reports its accuracy, i.e. probability for false positives and false negatives.

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 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. $\mathbf{I} = \langle i_0, ..., i_{|\mathbf{I}|-1} \rangle$, and sets where the order of their elements does not matter, e.g. $\mathbf{P} = \{p_0, ..., p_{|\mathbf{P}|-1}\}$. 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* $\{x^0, ..., x^d\} \in \mathbb{F}_2^{d+1}$ 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 $\{x^0, \ldots, x^{d-1}\} \in \mathbb{F}_2^d$ 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 $\{x^0, \ldots, x^d\}$, 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 $\{x_0, \ldots, x_{d-1}\}$ 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 $\{x_0, \ldots, x_{d-1}\}$. 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+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. In order 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 ISAs 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 $\{r_0, .., r_{14}\}$, the Program Counter (PC) (r_{15}) and the Program Status Register (PSR) (r_{16}). We refer to the list of all registers with $\mathcal{R} = \{r_0, ..., r_{16}\}$. 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 β based on a fixed effect size φ. 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 φ 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 φ) or to increase the reliability of the results (by decreasing p and β). 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[4]. 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

[4] https://github.com/emsec/arm-fault-simulator

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[5], 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[6].

4 Technique

We model each software, i.e., a compiled binary, as an ordered set of instructions $\mathbf{I} = \langle i_0, ..., i_{|\mathbf{I}|-1} \rangle$ 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 $\mathsf{type}(s) = \mathsf{reg}$ if s is a register index, $\mathsf{type}(s) = \mathsf{mem}$ if s is a RAM address, and $\mathsf{type}(s) = \mathsf{const}$ 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 A CPU-Independent Software Leakage Model

As software abstracts the underlying hardware's behavior, applying the robust d-probing model on the CPU design while executing a software would reduce the problem of verifying software to the formal verification of hardware. However, the ARM processor design, e.g. a hardware netlist, is most likely not available, which makes the hardware's formal verification impossible. This lack of information motivates the need for a CPU-independent adversary model used as a baseline for verifying masked software. 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 $\mathcal{R}^*$.

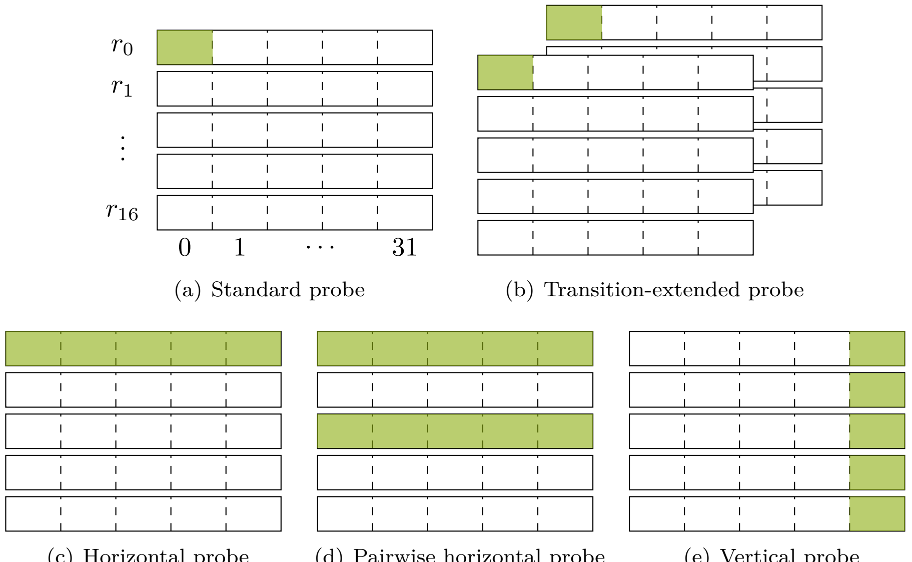
Definition 3 (Standard Probe)**.** A standard probe $p = (i, r, b)$ with $i \in \mathbf{I}$, $r \in \mathcal{R}^*$, and $b < 32$ is placed on the bit b of register r during instruction i.

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 the last instruction that modified r before executing i.

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

[5] https://salmanarif.bitbucket.io/visual/index.html
[6] https://www.unicorn-engine.org/



(a) Standard probe (b) Transition-extended probe

(c) Horizontal probe (d) Pairwise horizontal probe (e) Vertical probe

Figure 1: Different abstract probes on the registers.

Definition 6 (Horizontal Probe)**.** A horizontal probe $p_h = (i, r, \mathbf{B})$ with $i \in \mathbf{I}$, $r \in \mathcal{R}^*$, and $\mathbf{B} \subseteq [\![0, 32[\![$ 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 Modeling 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 why we restrict our model to worst-case evaluations with $\mathbf{R} = \mathcal{R}^*$ and $\mathbf{B} = [\![0, 31]\!]$. 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. Whenever the CPU interacts with a register, e.g., by reading or writing its state, 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+21] (see Figure 2). The resulting leakage is commonly referred to as neighbor leakage effect [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 neighbor leakage effect.

Proposition 1 (Model Neighbour Leakage Effects)**.** *A vertical probe $p_v = (i, \mathcal{R}, b)$ with $i \in \mathbf{I}$ and $b < 32$ covers all possible leakages sources by neighbour leakage effects on bit index b and during instruction i.*

To model all possible occurrences, i.e. for every instruction and bit index, we consider vertical probes on all $i \in \mathbf{I}$ and on all $b < 32$.

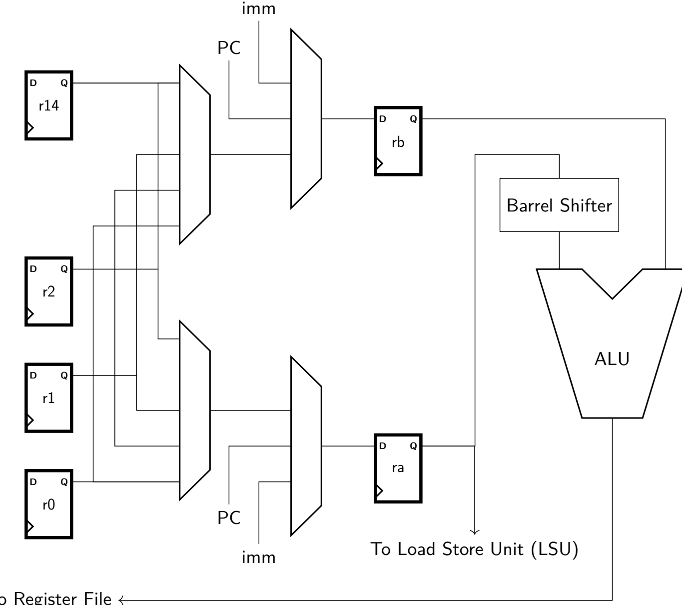


Figure 2: Simplified ARM Cortex-M3 pipeline structure [CGD18].

Bit-wise Interaction Leakage. 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 in probes on multiple bits of the operand register. Hence, this effect confirms Lemma 1 and indicates that horizontal probes model bit-wise interaction leakage.

Proposition 2 (Model Bit-wise Interaction Leakages)**.** *A horizontal probe $p_h = (i, r, [\![0, 31]\!])$ with $i \in \mathbf{I}$ and $r \in \mathcal{R}^*$ covers bit-wise interaction leakages in register r during instruction i.*

As the ALU processes two operands, the bit-wise interaction leakage is not bounded one register but to bits within both source registers of an instruction. To model bit-wise interaction leakage 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 $\mathbf{B} \subseteq [\![0, 31]\!]$ 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. 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 interaction with the RAM to model the multiplexers between r_{17} and the registers, as shown in [MPW22].

Memory Overwrite Effect. 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.

Proposition 3 (Model Memory Overwrite Effects)**.** *If instruction $i \in \mathbf{I}$ modifies $r \in \mathcal{R}^*$ it holds that a transition-extended probe $p = (\langle i, i' \rangle, r, b)$ with $i' \in \mathbf{I}$ and $b < 32$ covers*

all memory overwrite effects on register index b while i' denotes the last instruction that modified r before executing i.

Consequently, every aforementioned probe must become transition-extended in order to capture all possible memory overwrite effects. Further, we cover memory overwrites on the RAM by placing transition-extended probes on every targeted address of the RAM.

Memory Remnant Effect. While overwriting effects 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 memory remnant effect. 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 (Model Memory Remnant Effects)**.** *If instruction $i \in \mathbf{I}$ interacts with the RAM, i.e. reading the new state of $r \in \mathcal{R}$ from RAM or writing r to the RAM, it holds that a transition-extended probe $p = (\langle i, i' \rangle, r_{17}, b)$ with $i' \in \mathbf{I}$ and $b < 32$ covers all memory remnant effects on register index b 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}.

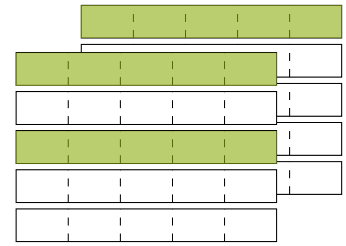
Pipeline Register Overwrites. If the CPU implements a pipeline, the integrated pipeline registers are prone to transitions combining operands of two consecutive instructions. Again, the Cortex-M3 implements a transition-prone pipeline (cf. Figure 2) [PV17]. Moreover, the pipeline effects of various ARM 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 (Model Pipeline Register Overwrites)**.** *A transition-extended probe $p = (\langle i, i' \rangle, r, b)$ with $i, i' \in \mathbf{I}$, $r \in \mathcal{R}$, and $b < 32$ covers all pipeline register overwrites on register r index b while i' denotes the last instruction that was executed right before i.*

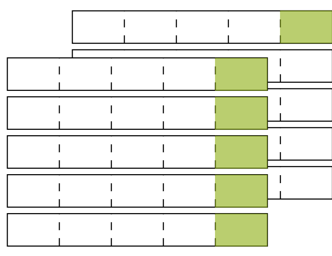
Pipeline register overwrites are already covered by vertical probes as both values (old and new) inside the pipeline are stored in their original registers as well.

4.1.2 Combined Occurrence of Glitches and Transitions

In contrast to [GD22], we consider glitches and transitions simultaneously and discuss their combined leakage. Therefore, we 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 an instruction, no transition extension is needed. For horizontal and vertical probes, we define their transition-extended variants as follows and visualize the concept in Figure 3.



(a) Transition-extended horizontal probe



(b) Transition-extended vertical probe

Figure 3: Different transition-extended probes on the registers.

Definition 9 ((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 $\mathbf{B} \subseteq [\![0, 31]\!]$ 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 9 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 10 (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 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 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 4 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 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.

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[7]. The compilation

[7] https://github.com/marketplace/actions/arm-none-eabi-gcc-gnu-arm-embedded-toolchain

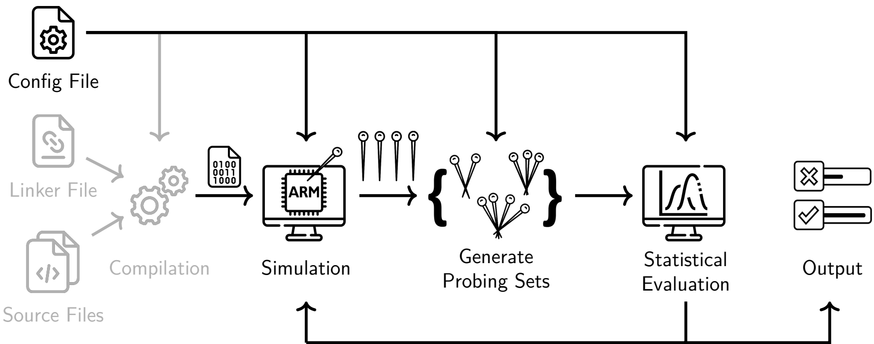


Figure 4: High-level overview of PROLEAD_SW.

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.

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

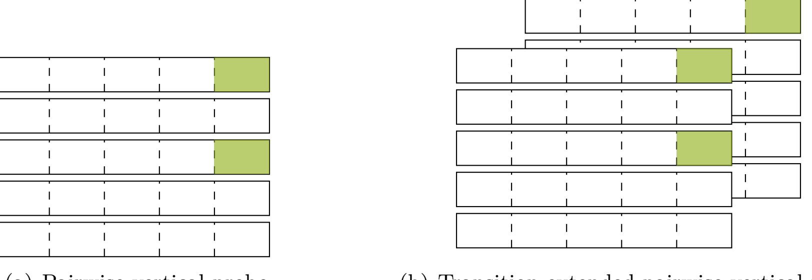


Figure 5: Pairwise vertical probes to improve efficiency.

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 5. 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 less 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 in RAM. If i writes its result in a register, we create standard probes on all bits of the destination register r, i.e., $p_0 = (i, r, 0), \ldots, p_{31} = (i, r, 31)$ 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 $p_h = (i, r, [\![0, 31]\!])$ (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.

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

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:  D* ← ∅
2:  for ∀d ∈ D do
3:      if type(d) = reg then
4:          r ← d                                   ▷ The corresponding target register
5:          D* ← D* ∪ {r}
6:          p_h ← (i, r, [[0,31]])                  ▷ Place a complete horizontal probe on register r
7:          P ← P ∪ {p_h}
8:          for ∀r' ∈ R\r do
9:              p_h ← (i, {r, r'}, [[0,31]])        ▷ Place a pairwise horizontal probe on register r
10:             P ← P ∪ {p_h}
11:     if type(d) = mem ∨ ∃s ∈ S s.t. type(s) = mem then
12:         D* ← D* ∪ {r_17}                        ▷ Add the hidden memory register
13: for ∀r ∈ D* do
14:     for ∀b ∈ [[0,31]] do
15:         p ← (i, r, b)                           ▷ Place a standard probe on bit b of register r
16:         P ← P ∪ {p}
17:         if r ≠ r_17 then
18:             p_v ← (i, R, b)                     ▷ Place a complete vertical probe on bit b
19:             P ← P ∪ {p_v}
20:             for ∀r' ∈ R\r do
21:                 p_v ← (i, {r, r'}, b)           ▷ Place a pairwise vertical probe on bit b
22:                 P ← P ∪ {p_v}
```

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+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 $p_h = (i, r_2, [\![0, 31]\!])$ observing all 32 bits of register r_2. After extension, the corresponding probing set contains 32 probes $p_0 = (i, r_2, 0), \ldots, p_{31} = (i, r_2, 31)$. The same applies for 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 $p_0 = (i, r_0, 5), \ldots, p_{16} = (i, r_{16}, 5)$.

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 $p_h = (\langle i, i' \rangle, r, [\![0, 31]\!])$ instead of $p_h = (i, r, [\![0, 31]\!])$ 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 for 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 $\mathcal{G}$ which 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. 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 `0xda39a3ee5e6b4b0d3255bfef95601890`, 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+22], we restrict the number of simulations to 10 000.

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+22]. The underlying masking schemes differ across the implementations. For their experimental analysis, the authors of [BWG+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[8]. 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, 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[9].

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 transitional leakage 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
```

[8] https://github.com/coron/htable
[9] https://github.com/coron/htable

```
    [...]
```

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[10], 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+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.

```
<multshare_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+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. 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+21] combined IPM and fault detection techniques to create a masked AES implementation[11], 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 $L = (l_1, \ldots, l_n)$ and $R = (r_1, \ldots, r_n)$ are constructed, such that x is the 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 using normal probes, i.e., no transitional, horizontal, or vertical. 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);
```

[10] https://github.com/coron/htable
[11] https://github.com/Qomo-CHENG/IPM-FD

```
        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[12] made use of affine masking [FMPR10] to create a first-order protected AES implementation available via GitHub[13], 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 Rinjdael 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 unmasks 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 with all effects enabled did not show any leakage for 10 000 simulations. 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.

[12] https://www.ssi.gouv.fr/
[13] https://github.com/ANSSI-FR/SecAESSTM32

We used the dedicated test cases, provided by the authors on GitHub[14] for the evaluation of all three gadgets used in their K3 and S3 implementation of Kyber [BDK+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 only normal probes. 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. 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, even if we consider all effects (vertial and horizontal probes), 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 modulo p.

Results. As before, PROLEAD_SW detects no leakage even considering transitional probes on registers or memory read and writes. However, contrary to the experimental results shown in [BC22], PROLEAD_SW detects first-order leakage after a few hundred simulations if vertical probes are covered. We used PROLEAD_SW's output to trace the 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 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.

[14] https://github.com/uclcrypto/pqm4_masked

```
<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 enable all effects except vertical probes, PROLEAD_SW detects no leakages after simulating 10 000 runs. However, when we enable vertical probes, 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[15]. We evaluated all schemes with 10 000 simulations and enabled transitions, horizontal probes and vertical probes. 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.

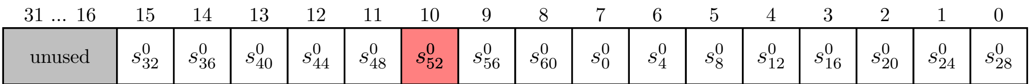
[15] https://github.com/KULeuven-COSIC/Software-masked-Keccak-and-PRESENT

5.5.1 PRESENT

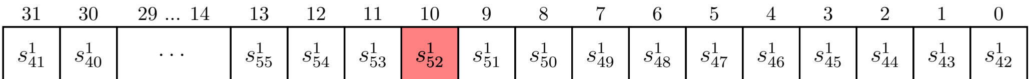
The masked cipher takes two shares of the 64-bit plaintext as inputs. In order to ensure register non-completeness, the authors have shifted one share of the sensitive values one bit to the right. Then, the permutation layer of PRESENT is performed share-wise. As the non-linear layer contributes to the highest overheard 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
    [...]
```



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^1_{32} at the most significant bit position, followed by s^1_{63}, s^1_{62}, s^1_{61} and so forth until we end up with s^1_{33} 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.



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.

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+21, CHK+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 enables, its evaluation tends to be conservative as it covers a worst-case scenario, 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 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

Table 1: Evaluation results. **Effects** column indicates the first configuration with which we detected leakages. Effects are ordered and abbreviated as N=normal probe, T=transition, M=memory, H=horizontal, V=vertical.

Design	Effects	Security	#Instr	Performance		
		(order, achieved)		[#sets]	[RAM]	[time]
AES, RP [RP10]	T	(1, ✗)	11 520	741 856	5.5 GB	14 sec
AES, Htable [Cor14]	T	(1, ✗)	71 033	4 168 384	9.8 GB	1 min
AES, IPM [CCG+21]	N	(1, ✗)	113 301	8 122 880	13 GB	35 min
AES, FLR [CGZ20]	N	(1, ✗)	25 208	1 567 680	5.8 GB	22 sec
ANSSI AES* [FMPR10]		(1, ✓)	2 854	358 096	21 GB	17 min
SecAdd [BC22]		(1, ✓)	4 024	3 808 448	18.3 GB	27 min
SecB2AModp [BC22]	V	(1, ✗)	variable†	89 686 816	123 GB	21 h
SecA2BModp [BC22]	V	(1, ✗)	9 852	9 648 224	45 GB	10 min
PRESENT [GD22]	V	(1, ✗)	2 799	1 401 856	14 GB	4 min
Keccak-f1600 [GD22]	V	(1, ✓)	3 830	318 912	95 GB	29 min
Keccak-f800 [GD22]	V	(1, ✓)	2 035	160 622	45 GB	14 min
Quadratic Classes [GD22]	V	(1, ✓)	370	29 726	8 GB	3 min

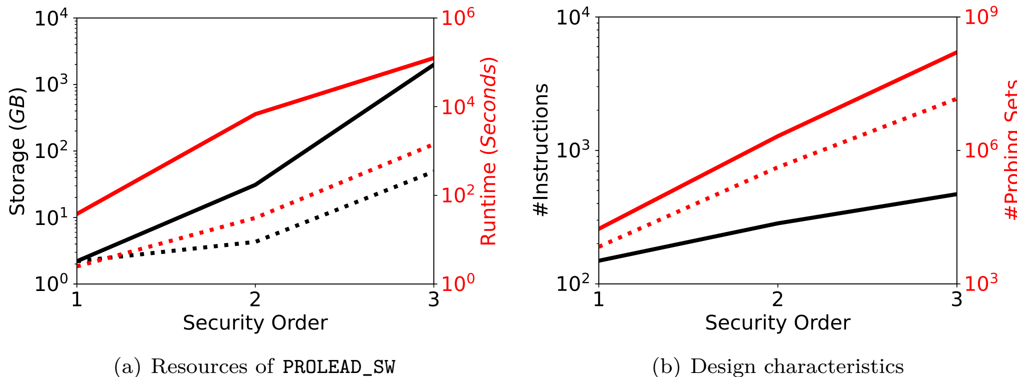
* specific GitHub implementation[13] † on average approximately 50 000 instructions

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[16], 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 simlulations. The benchmarking results, visualized in Figure 6, are twofold. First, we conducted the evaluations without considering any micro-architectural effects (indicated by dotted lines). Then, we enabled the detection of any covered micro-architectural effect and reran the evaluations (indicated by solid lines).



(a) Resources of PROLEAD_SW (b) Design characteristics

Figure 6: 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 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.

[16] https://github.com/uclcrypto/pqm4_masked

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. 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 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+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+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+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+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+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède 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+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+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+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+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+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih 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.

[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+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+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.

[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+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+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, accessed on Jan. 2023.