aLEAKator: HDL Mixed-Domain Simulation for Masked Hardware & Software Formal Verification

Noé Amiot[1], Quentin Meunier[1], Karine Heydemann[1,2] and Emmanuelle Encrenaz[1]

[1] Sorbonne Université, LIP6, CNRS, 4 Place Jussieu 75005, Paris, France, first.last@lip6.fr
[2] Thales, Meyreuil, France, first.last@thalesgroup.com

Abstract. Verifying the security of masked hardware and software implementations, under advanced leakage models, remains a significant challenge, especially when accounting for glitches, transitions and CPU micro-architectural specifics. Existing verification approaches are either restricted to small hardware gadgets, small programs on CPUs such as Sboxes, limited leakage models, or require hardware-specific prior knowledge.
In this work, we present aLEAKator, an open-source framework for the automated formal verification of masked cryptographic accelerators and software running on CPUs from their HDL descriptions. Our method introduces mixed-domain simulation, enabling precise modeling and verification under various (including robust and relaxed) 1-probing leakage models, and supports variable signal granularity without being restricted to 1-bit wires. aLEAKator also supports verification in the presence of lookup tables, and does not require prior knowledge of the target CPU architecture. Our approach is validated against existing tools and real-world measurements while providing innovative results such as the verification of a full, first-order masked AES on various CPUs.

Keywords: Side-Channel Attacks · Masking · Micro-Architectural Leakage Detection · Formal Verification

1 Introduction

1.1 Context

Side-channel attacks, discovered in the late 90's [Koc96, KJJ99], are still a major threat for hardware and software implementations of cryptographic algorithms. These attacks exploit physical quantities such as power consumption, electromagnetic emissions, or timing information to retrieve secret data manipulated by a cryptographic system.

Introduced in 1999 [GP99, CJRR99], the *masking* countermeasure has been formalised by Ishai *et al.* [ISW03]. At order d, it consists in splitting a secret variable s into $d+1$ parts $s_0 \dots s_d$, called *shares*, using d uniformly distributed random values and such that $s = s_0 \star \cdots \star s_d$ where $\star$ is the operator used for sharing. For example, Boolean masking uses the $\oplus$ operator, whereas Arithmetic masking uses the arithmetic addition $+$. With such a sharing, any combination of d or less shares is statistically independent of the secret variable s. Since the seminal Boolean masking scheme [ISW03], other schemes such as Threshold Implementation (TI) [NRR06] and Domain-Oriented Masking (DOM) [GMK16] have been proposed to either eliminate the randomness needs, resist glitches or even ease the secure composability of masked circuits.

A masked implementation can be proven secure according to a *leakage model* which defines the information an attacker can retrieve while observing the circuit or the running program. In the `d-probing model` [ISW03], an attacker is able to observe any `d`-uplet of intermediate values computed by the implementation. An implementation is said `d-probing secure` if any `d`-uplet is statistically independent of secret data. Yet, this rather simple and widely used model does not always match real leakages, as other physical effects contribute to the dissipated power consumption or EM emanations. Due to the propagation delay of signals through gates and wires, the input wires of a gate may stabilise at different instants or change several times before stabilising, hence inducing a potential transient toggling of the gate output, named *glitch*, which possibly reveals information on the inputs of the gate. When using the CMOS technology, significant instantaneous power consumption occurs when a wire changes from a logic value 0 to a logic value 1 or conversely [MOP07]. As a consequence, an attacker is able to observe the *transitions* between consecutive values of a wire with a single probe. Extended leakage models have been introduced to cover these hardware effects. The `robust d-probing model` [FGM+18] extends the `d-probing model` with either transitions, glitches, or both, and is expressed as `(g,t) d-probing model`[1], in which `g` and `t` are set to 0 or 1 accordingly.

In order to reduce the set of considered glitches only to those that may occur in practice, the notion of signal *stability* was recently introduced when glitches cannot occur [HB21, MM24]. Specifically, the stability of a wire indicates whether its value has changed during the current cycle. For a register, its output is stable if and only if its value is the same as in the previous cycle. For a combinatorial gate, its output is stable when its whole input combinatorial dependency tree does not change value in the current cycle. As stable wires cannot induce glitches, a probe on a stable wire only leaks the wire value for the current cycle. The `Robust but Relaxed d-probing model` [MM24] or `RR d-probing model` is defined as the `(1,1) d-probing model` while considering stability of the signals.

1.2 Open Research Questions

Various formal approaches have been proposed to prove whether a masked hardware or software implementation leaks within a specified leakage model. Verifying the absence of secret leakage in a software masked implementation requires to consider the target device micro-architecture that dictates the potential source of leakages hence reducing the security of the masked software [MPW22, BWG+22, GPM21]. A potential source of leakage causes a leak depending on the code being executed, advocating for a co-verification of the hardware and the software. Moreover, if the composition of formally verified masked gadgets is a possible approach for producing a fully masked implementation, it requires formally proving the security of gadget composition. For hardware implementations, this is already covered by several works proposing security properties covering secure composition such as Strong Non-Interference (SNI) [BBD+16] and Probe Isolating Non-Interference (PINI) [CS20], as well as methods for formally proving them [BBC+19, KSM20, ZCF24]. Regarding software, secure composition of gadgets is still an open question, in addition to the fact that masking schemes dedicated to a specific algorithm are more likely to be deployed, as they most often simultaneously reduce the required amount of randomness, the execution time, and the code size w.r.t. a gadget-based masked implementation. Today, state-of-the-art masked software verification solutions are however limited to 1) either small programs, typically software gadgets, with a limited number of executed cycles while possibly considering the most powerful leakage model [GHP+21, GPM21]; 2) fully masked programs while only considering `(0,1) 1-probing leakage model` and a manually extracted model from the CPU design [GHM22], potentially leading to missed leakage sources; 3) full program verification using a 2-step approach based on contracts [HHB24]. Contracts must be verified once against the hardware target, then the verification

[1] We omit the parameter `c` for the coupling effects, not accounted for in the following.

of the software is performed considering the verified contracts. Such power contracts are manually defined, necessitating advanced prior knowledge of the target device.

As some optimised masked implementations make use of lookup tables to retrieve precomputed results, e.g. [HOM06], verification techniques should be able to support such masking optimisations. Verifying a masked software running on a CPU model resulting from the synthesis step — i.e. once the mapping with the target technology has been carried out — allows to only detect leakages visible after this mapping but it has two drawbacks. First, the verification output is technology-dependent; second, the verification is necessarily carried out at the bit level [GHP+21]. At this level, it is difficult to identify all the bits composing a wire after the synthesis. It is therefore not possible to correctly verify masked implementations deployed using n-bit shares because it becomes difficult to recombine n-bit wide intermediate values, when n is greater that one.

As a result, overcoming the limitations identified in current verification approaches remains an open challenge. A key objective is to enable the automatic verification of larger programs running on a wide range of CPUs, under strong leakage models. In particular, achieving verification without requiring prior knowledge of the target CPU architecture, and without restricting the analysis to 1-bit wires, is a crucial property of future verification techniques.

1.3 Contributions

To address the aforementioned challenges, we propose a novel methodology, implemented as a dedicated tool, that enables the verification of the absence of secret leakage in both cryptographic hardware accelerators and masked software executing on CPUs.

Specifically, we introduce and formally define a new simulation technique called *mixed-domain* simulation. This approach allows the comprehensive collection of all necessary information to perform verification within a range of leakage models, including the `RR 1-probing model`. Furthermore, we systematically identify and specify the set of wires that require verification for each supported leakage model. This targeted selection significantly reduces the number of verifications required to assess a property, without compromising security: any potential leakage from a non-verified signal necessarily propagates to a verified signal.

We present aLEAKator, implementing the mixed-domain simulation technique and verification requests optimisation. aLEAKator is a comprehensive framework for the automated formal verification of cryptographic accelerators and programs running on CPUs: It processes the HDL code of a masked hardware or of a system-on-chip embedding a CPU to extract a model representative of post-synthesis designs, preserving original signal widths and that can be simulated. This enables for the verification at varying levels of granularity, whether at the bit or signal level. The framework supports the specification of tailored verification rules for masked programs involving lookup table accesses. Notably, aLEAKator is open source[2], with all necessary tooling for model extraction and verification.

To demonstrate the versatility, efficiency, and accuracy of aLEAKator, we conduct extensive experiments. First, we validate our method by successfully reproducing eight state-of-the-art formal verifications of cryptographic hardware, achieving up to a 50-fold speed-up over previous approaches. Next, we assess the security of both the original and a hardened version of a masked gadget, as well as two complete applications, in the `1-probing model` and `RR 1-probing model`. Verification is performed on five distinct CPUs (Coco-Ibex, Ibex, CV32E40P, Cortex-M3, Cortex-M4). To our knowledge, this marks the first formal verification in the `RR 1-probing model` of a full first-order masked AES implementation running across several CPUs, with aLEAKator completing verification in only 19 to 39 minutes, depending on the core. Finally, we compare our verification

[2] Available at https://github.com/noeamiot/aLEAKator

findings against eight real power measurements on two Arm CPUs. We observe that both original versions, flagged by aLEAKator as leaky, do exhibit leakage in practice, whereas the hardened versions, certified as secure by our tool, show no practical leakage. This strongly supports the effectiveness of our approach for formal leakage verification.

All benchmarks—including hardware accelerators, masked programs, and CPU models (excluding the ARM cores)—are made available alongside aLEAKator.

In the following, we first go through the background key verification notions and existing works (Section 2) before explaining our method and its formal definitions (Section 3). After a presentation of the implementation (Section 4), we present experimental results and discuss our approach (Section 5) before finally concluding on our work (Section 6).

2 Background on Verification of Masked Implementations

Various approaches have been proposed to assess the security of masked software implementations. Two main types of verification coexist: statistical approaches, which rely on statistical analysis applied to simulated or real traces, and formal approaches, which aim to verify the absence of leakage during program execution w.r.t. a given leakage model.

2.1 Verification Based on Statistical Analysis

The statistical analysis based approaches have been studied in various works within the previous decade.

`ELMO` [MWO16] is an instruction level power simulator which can generate power traces for the ARM Cortex-M0 and Cortex-M4 processors. Instruction power consumption is estimated using linear regression on real measurements per instruction. `ELMO*` [SSB+21] is an enhanced version of `ELMO` which accounts notably for non-consecutive instructions and memory accesses. This simulator can then be used for fast verification using statistical metrics such as the *t*-test or the SNR. It is integrated in `ROSITA` [SSB+21] and `ROSITA++` [SCS+21], two code rewrite engines able to secure respectively first-order masked and higher-order masked implementations by exploiting `ELMO*` outputs.

`MAPS` [CGD18] is a micro-architectural power simulator for the Cortex-M3. The Cortex-M3 model includes only the registers involved in the manipulation of data, identified in the HDL description. `MAPS` simulates ARM binaries to help identify leakages, in particular those introduced by the pipeline registers.

`Prolead` [MM22] is a verifier based on simulations and statistical hypothesis tests. By simulating the netlist of a circuit with numerous input vectors while placing probes at key locations, `Prolead` computes statistical independence with the G-test and is able to verify circuits in the `RR d-probing` model. As it is based on concrete simulations, the certainty of the results grows with the number of tested input vectors. `Prolead` can be used on large stateful (i.e. in which inputs are not fixed over time) circuits.

`Prolead_sw` [ZMM23] extends `Prolead` to support the verification of masked software by simulating the execution of binaries running on a generic CPU model. It uses a CPU-independent abstract leakage model which encompasses various leakage sources reported in state-of-the-art works. It currently only supports ARM binaries.

2.2 Formal Verification Methods

Formal verification approaches, to prove whether a masked hardware or software implementation leaks within a defined leakage model, are usually split into two steps.

The first step consists in building a model of the target circuit or the target CPU running the masked software, representing all possible leakage sources in the chosen leakage model. Any input of the circuit or the software must be typed or labeled as *public* for unsensitive data, and either *masks* and *secrets* or *shares* for sensitive data. From this model, a symbolic intermediate representation of all the data, sensitive or not, propagating through the circuit (or CPU) is extracted.

The second step is the application of a verification method to prove that the extracted symbolic intermediate representation of propagating data cannot leak information about the secrets in a given leakage model for all possible inputs. We identified three main verification methods. 1) The *substitution*-based methods rely on symbolic expressions to represent the intermediate values. They perform successive replacements of masked sub-expressions with a bijective mask, and aim at removing all secret variables from the expression, in which case the expression is statistically independent from all secrets. Introduced by [BBD$^+$15], this method is implemented in all the versions and variants of `maskVerif` [BBC$^+$19] as well as in `VerifMSI` [MT23] and its previous version [MPH23]. 2) The *inference*-based methods also use symbolic expressions, along with types for describing different distributions of an expression, and rules to infer the distribution type of an expression from the types of its sub-expressions. Initially proposed by [BEOMHE19] for verifying first order masked binary code, it has been extended with finer rules and to support more complex code structures [GXZ$^+$19]. 3) Finally, verification methods based on *constraints satisfiability* either express the absence of leakage in intermediate values with several constraints given to a SAT or SMT solver as in `Coco` [HB21]; or represent intermediate values with specific structures such as ROBDD to efficiently determine their distribution, e.g. in the tools `SILVER` [KSM20] and `Prover` [ZCF24].

All mentioned formal verification methods except those based on ROBDD may have false positive. To discriminate between true and false positives, one has to determine the statistical independence of an expression w.r.t the secret values as presented in [EWS14], Figure 1. A resort consists in enumerating all possible values for symbolic inputs to compute this statistical independence. Note however, that this is not possible for large symbolic input domains.

In Section 2.4, we review existing work on the formal verification of masked software that takes into account micro-architectural leakage sources of the target CPU. In the following section, we discuss the impact of verification granularity, as previously introduced.

2.3 Probe Width in Probing Models

Since the earliest work on provable masking [ISW03], pen-and-paper verifications of generic masking schemes [ISW03, RP10] have been considering a `d-probing` model, in which probes have an implicit width defined by the analysed description. In particular, it is 1-bit for masked circuit expressed at gate level [ISW03], and the width of the intermediate computations for masked algorithm, e.g. n-bit for the masked multiplication over $\mathbb{F}_{2^n}$ algorithm or 8-bit for the masked AES [RP10].

Automated verification methods presented in the previous section mainly target one kind of implementation, either hardware or software, at a specific level such as RTL or gate level for hardware, and such as assembly or pseudo-code level for software. In these verification methods, the probe width is always implicit: 1-bit for the hardware [BBC$^+$19, KSM20, ZCF24] and 8-bit for the software [BEOMHE19, GXZ$^+$19].

When it comes to the verification of a program running on a CPU while considering its micro-architecture, one question arises: what should be the width of the probes? This also questions the probe width to consider when verifying hardware.

The probe width defines which signals or variables can be seen “all at once”. The issue regarding masked software is that the masking scheme support width can be different from the one carrying the information in hardware or at a lower level. For example, `int8_t` variables in `C` will be manipulated on 32-bits registers and wires on a 32-bit ISA, possibly leading to more than one variable on a wire or register, depending on the compiler optimisations. Moreover, the probe width can seem totally arbitrary from a hardware perspective as in the final circuit wire or register bits are not necessarily close to each other. However, the probe width has a real impact on the security guarantee delivered by the verification. Considering only 1-bit signals implies that the eight bits of a byte wire can all be correctly masked with the same random bit; e.g. a masked design using only the two masks 0x00 and 0xFF would be proven secure if only 1-bit signals are verified, but would obviously not be proven secure at byte level, i.e. when considering 8-bit signals. Therefore, the verified security property is not the same whether considering 1-bit or wider signals, and wider signals verification brings a stronger security guarantee.

First, for clarity and reproducibility, we argue that the probe width should be made explicit in any formal verification setting. Second, from a security perspective — particularly in software verification based on hardware descriptions or at the ISA level — we advocate taking into account the width of the hardware elements that carry or store sensitive data. In the context of hardware verification, if the goal is to verify a masking scheme under the assumption that each bit of a wire leaks independently from the others, then verification can be performed on a per-bit basis. Conversely, to detect violations of this independence assumption or to support masking schemes where all bits of a wire are assumed to leak jointly, verification must be carried out on the fully reconstructed wire. To explicitly capture the probe width and distinguish between these two verification paradigms, we propose to refine the `d-probing` leakage model as either the `d-bit-probing` model or a support-wise leakage model: the `d-sw-probing` model. In the following, the `d-sw-probing` model is always considered with stability, as in the `RR d-probing` leakage model.

A limitation of the `d-sw-probing` model is that it depends on the hardware design choices, e.g. a processor could be described in HDL using only 1-bit signals. However, we expect that most of the time, the signal definitions in HDL are relevant w.r.t. the masking scheme and the desired leakage model.

2.4 Software Verification while Considering CPU Micro-Architecture

In this section, we present the existing approaches and associated tools for the formal verification of masked software running on CPUs, while considering the micro-architecture of the CPU. These approaches are aggregated in Table 1.

Table 1: Comparison of formal verification methods of software running on CPU.

Feature/Tool	Armistice	Coco	Closing the gap	aLEAKator
Multi-bit signals	✓	✗	✓	✓
Probing model	`(g=0,t=1) 1-sw`	`RR d-bit`	`RR d-bit`	`RR 1-sw`
Cryptographic accelerators	✗	✓	✗	✓
Automatic from HDL	✗	✓	✗[3]	✓
Usable CPU(s)	Generic CPU model	Coco-Ibex, Ibex, SweRV	Ibex	Cortex-M3, Cortex-M4 Ibex, Coco-Ibex, CV32E40P
Memory model	✓	✓	✗	✓

`Armistice` is devoted to the simulation of a model of the Cortex-M3 datapath in order to verify masked software implementations running on it to cover leakages stemming from micro-architectural features. The model have been manually created from the

[3] The contract redaction is a manual CPU-dependent task that must be performed before being verified.

Verilog description of the Cortex-M3, which is a highly non-portable and error-prone task, advocating for an automated extraction as we propose. Besides, `Armistice` only supports value-based and transition-based leakage models. Still, `Armistice` handles multi-bit signals as defined by the manually modelled Cortex-M3 and allows for support-wise verification.

`Coco` [HB21] is a SAT-based formal verifier that can be used on programs running on CPU as well as hardware accelerators. It has been used on the RISC-V Ibex and SweRV CPUs [GHP+21, GPM21]. From an input labelling and a HDL description, it drives a simulation to extract from the simulation trace, and a model of the hardware target, the intermediate data to verify while considering transitions, glitches as well as stability. `Coco` only performs bit-probing verification.

`Closing the gap` [HHB24] is a contract-based formal method only for masked software running a CPU. Contracts express, at the ISA level, all the potential leakages. They are first validated against the HDL CPU description to ensure they capture all the potential leakages. Subsequently, masked programs are verified using symbolic simulation and the contracts. The main advantage of this approach is that the verification of the hardware against the contract, while being computation-intensive, is only done once. Afterward, the verification of the program is only performed w.r.t. the contract instead of a CPU model, thereby alleviating the burden of the verification. This approach requires extensive knowledge on the CPU to manually establish the contract. Currently, only the Ibex CPU is supported, and memories are not accounted for. The method claims to handle "word-wise" verification but does not describe how the bits are gathered into words, and the authors only justify word granularity for performance reasons. Therefore, verifications are not performed considering a support-wise probing model.

`aLEAKator` is the implementation of a new method combining a *mixed-domain simulation* for the modelling of the circuit, and a substitution-based approach for the verification step. Its versatility allows for verification in variations of the `RR 1-sw-probing` model of both masked hardware and masked software running on several CPUs. The HDL description of the circuits and micro-architectures are directly exploited to generate intermediate representations of the models without particular knowledge on the verified target. It handles multi-bit signals as defined in Section 2.3 and accounts for the micro-architectural leakage sources in the memory subsystem.

3 Method

3.1 Overview

Traditional HDL simulation computes values in the Boolean domain for all signals at each cycle. We refer to mixed-domain simulation as a simulation extension, by enriching the signal with values in other domains. This allows for the verification of secret leakage absence in a circuit, in presence of transitions and glitches. The additional signal values include:

- Symbolic expressions that describe wire values with respect to sensitive information such as shares, secrets, and masks.
- Sets of symbolic expressions, referred to as *LeakSets*, which represent all possible observable values when considering glitches.
- Stability information, for the eventual refinement of *LeakSets*, needed for the `RR 1-sw-probing` model.

Traditional concrete values are also computed, in particular for validation, allowing to check the correctness of the generated symbolic expressions. The simulation process thus occurs across four domains: the concrete domain, the symbolic domain, the *LeakSet* domain, and the stability domain.

The flow of the method is depicted in Figure 1. The first input is a circuit which is either a CPU or a cryptographic accelerator, given in a Hardware Description Language (HDL). In the case of a processor, a binary program must also be provided. The other inputs are a leakage model, up to the `RR 1-sw-probing` model, stimuli needed to simulate the circuit such as clock or reset signals for all cycles and labels that maps the symbolic variables to their type, being either *share*, *secret*, *mask* or *public*.

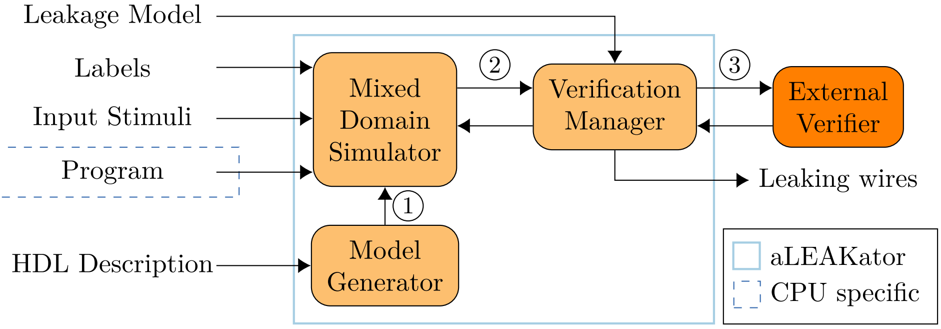


Figure 1: Overview of aLEAKator.

The flow begins with the generation, from the HDL description, of a model (step 1) that can be simulated while exposing all intermediate values needed for verification. This model is subsequently simulated (step 2) using user-provided inputs stimuli and labels. After each simulation cycle, the generated intermediate values are used by the verification manager which handles the interface with the external verifier, providing it with the relevant combinations of elements to verify in the specified leakage model (step 3). Reported leaks are output with the simulation cycle number, their description (i.e., which symbolic expression or *LeakSets*), their location (HDL source file, line number and wire name). This precise information can greatly help the designer understand and fix the reported secret leakages. The simulation (step 2) and verification (step 3) sequence iterates until the sequence of input stimuli have been entirely processed.

In the remainder of this section, we formally define the circuit model we consider, the different domains and the simulation in each domain.

3.2 Circuit Modelling

Hardware accelerators and CPUs running software are sequential circuits modelled in HDL and whose behaviour is modelled as functional Mealy machine. CPU targets generally embed data and instruction memories, which are here considered as part of the circuit.

Definition 1 (Circuit)**.** A circuit $\mathcal{C}$ is described with the tuple $\langle W, G, init \rangle$ where G is the set of gates, which are either combinatorial ($\in C$) or registers ($\in R$) and *init* is the vector of initial values for R. W is the set of wires and contains I, the set of primary inputs, O, the set of primary outputs and other internal wires ($\in G \times G$). Any $w \in W$ has a width denoted $|w|$.

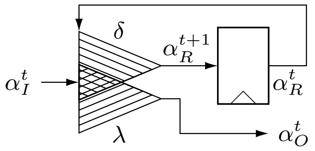
Definition 2 (Combinatorial gate)**.** A combinatorial gate $g \in C$ is defined by the tuple $\langle f_g, W_g^{in}, w_g^{out} \rangle$ where f_g is its functionality, W_g^{in} is the set of its input wires and w_g^{out} is its unique output wire. It can have multiple inputs but has a single output.

Definition 3 (Sequential gate)**.** A sequential gate a.k.a register, is a gate r defined by the tuple $\langle w_r^{in}, w_r^{out} \rangle$ where w_r^{in} and w_r^{out} are its unique input and output wires. Sequential gates encompass memory elements.

Definition 4 (Functional Mealy machine)**.** The functional Mealy machine $\mathcal{M}(\mathcal{C})$ associated with a circuit $\mathcal{C}$ is described with the tuple $\langle \Sigma_I, \Sigma_O, \Sigma_R, \delta, \lambda, \alpha_R^0 \rangle$ with Σ_I, Σ_O, Σ_R being respectively the sets of all valuations on I, O and output wires of elements in R, this latest also denoted as the set of states.

In Definition 4, $\alpha_R^0 \in \Sigma_R$ is the initial state of $\mathcal{M}(\mathcal{C})$, it corresponds to the *init* vector of $\mathcal{C}$. More generally, we denote $\alpha^t \in \Sigma_W$, the valuation of all wires in $\mathcal{C}$ at cycle t, and α_X^t the restriction of α^t to the wire or set of wires X. In particular, α_R^t is called the state of the circuit at cycle t; α_I^t and α_O^t are the input and output valuations at cycle t. A representation of a functional Mealy machine is given in Figure 2.

In Definition 4, δ: $\Sigma_I \times \Sigma_R \to \Sigma_R$ is the transition function and λ: $\Sigma_I \times \Sigma_R \to \Sigma_O$ is the output function. Their application computes respectively the state for the next cycle and the current output valuations from the current input and current state, as given by Equation 1.



$$\forall t \geq 0 \begin{cases} \alpha_R^{t+1} = \delta(\alpha_R^t, \alpha_I^t) \\ \alpha_O^t = \lambda(\alpha_R^t, \alpha_I^t) \end{cases} \tag{1}$$

Figure 2: Functional Mealy machine.

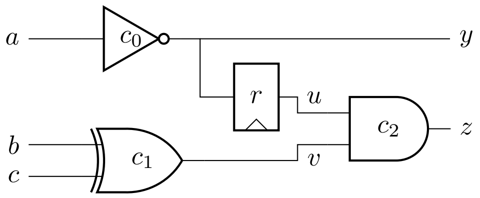
Note that δ and λ are invariant in time as the structure of the circuit is fixed. Moreover, as they compute a valuation for each element in R and O respectively, they both are vector-valued functions, composed of component functions, one for each element (of R or O) to compute.

The δ and λ functions are built by applying the inductive traversal of the structure of the circuit $\mathcal{C}$ (Algorithm 1) on the input wires of each register (Equation 2) and on each primary output (Equation 3) respectively.

$$\delta = \left| \begin{array}{l} \textsc{build}(w_{r_0}^{in}) \\ \vdots \\ \textsc{build}(w_{r_n}^{in}) \end{array} \right. \quad r_0 \ldots r_n \in R \tag{2}$$

$$\lambda = \left| \begin{array}{l} \textsc{build}(w_0) \\ \vdots \\ \textsc{build}(w_n) \end{array} \right. \quad w_0 \ldots w_n \in O \tag{3}$$

Figure 3b provides the δ (Equation 4) and λ (Equation 5) functions of Mealy machine associated to the circuit example given in Figure 3a and obtained using Equations 2 and 3 respectively. This circuit contains both combinatorial and sequential gates.



(a) Structural view.

$$\delta = \left| \; \delta_r \equiv f_{c_0}(\alpha_a) \right. \tag{4}$$

$$\lambda = \left| \begin{array}{lll} \lambda_y & \equiv f_{c_0}(\alpha_a) & \\ \lambda_z & \equiv f_{c_2}(\alpha_u, & \alpha_v \quad) \\ & \equiv f_{c_2}(\alpha_r, & f_{c_1}(\alpha_b, \alpha_c) \quad) \end{array} \right. \tag{5}$$

(b) Corresponding δ and λ functions.

Figure 3: (a) A circuit example with $I = \{a, b, c\}$, $O = \{y, z\}$ and internal the wires $\{u, v\}$. Gates sets are defined as $C = \{c_0, c_1, c_2\}$ and $R = \{r\}$. In this example, f_{c_0} is a NOT functionality, f_{c_1} a XOR and f_{c_2} a AND. (b) The transition and output functions of the functional Mealy machine $\mathcal{M}(\mathcal{C})$.

Algorithm 1: Inductive traversal of the structure of the circuit $\mathcal{C}$ from a wire w.

```
Input  : A wire $w$, which is either a primary input or the output wire of a gate $g$
Output : The functionality associated with $w$ up to the inputs and registers
BUILD($w$):
    if $w \in I$ then return $w$
    else if $g \in R$ then return $w_g^{out}$
    else return $f_g(\ldots,$BUILD$(w_i),\ldots)$            ▷ With $w_i$, the wires in $W_g^{in}$
```

Starting from $\alpha_R^0 \in \Sigma_R$, the Mealy machine, driven by the input configuration sequence $\alpha_I^0 \ldots \alpha_I^{n-1} \ldots$, generates, through successive applications of λ and resp. δ, a sequence of states $\alpha_R^1 \ldots \alpha_R^n \ldots$ and resp. a sequence of output configurations $\alpha_O^0 \ldots \alpha_O^{n-1} \ldots$. These intertwined sequences are represented as follows:

$$\alpha_R^0 \xrightarrow[\alpha_O^0]{\alpha_I^0} \alpha_R^1 \rightarrow \ldots \rightarrow \alpha_R^{n-1} \xrightarrow[\alpha_O^{n-1}]{\alpha_I^{n-1}} \alpha_R^n \xrightarrow[\cdots]{\cdots} \ldots$$

3.3 Mixed Domain Simulation

One application of the δ and λ functions associated to $\mathcal{C}$ represents the simulation of the circuit during one clock cycle, while the successive applications of these functions simulate $\mathcal{C}$ over several clock cycles. The usual simulation of a circuit is a simulation in the *concrete* domain. It consists in computing logical values on each wire.

Security properties evaluation is based on the analysis of interaction of all sensitive data during computation over time. We introduce symbolic simulation to capture the exact contribution of each sensitive data to all intermediate computations, at each spatial and temporal point. The mixed-domain simulation extends the concrete simulation by propagating symbolic expressions representing all calculus w.r.t. the sensitive data, leading to a simulation in the *symbolic* domain. The set of values which can be observed when considering glitches is propagated in the *LeakSet* domain and the information on stable signals is made in the *stability* domain, to eventually refine the *LeakSet* computation.

We formally define these domains to represent the propagated values, along with the behaviour of each gate type in each domain, using dedicated functionality functions.

1. **Concrete Domain**: $\mathcal{B}$, the set of Boolean vectors of all sizes, $\mathcal{B} = \bigcup_{i \in \mathbb{N}^+} \mathbb{B}^i$, with $\mathbb{B} = \{0, 1\}$.
2. **Symbolic Domain**: $\mathcal{E}$, the set of symbolic expressions of all widths, $\mathcal{E} = \bigcup_{i \in \mathbb{N}^+} \mathbb{E}^i$, with $\mathbb{E}^i$ the set of all expressions e of width i defined as follows:

$$e \in \mathbb{E}^i \Leftrightarrow e := \begin{cases} \texttt{CST}(x) & \text{a constant } x \in \mathbb{B}^i \\ \texttt{SYMB}(y) & y\text{, a symbolic variable of width } i \\ \texttt{OP\_}*(\ldots, t_j, \ldots) & \texttt{OP\_}*\text{, an operator of output width } i \text{ and } t_j \in \mathcal{E} \end{cases} \tag{6}$$

Table 2 gives the complete list of operators which can appear as symbolic expression constructors.

Table 2: List of function operators in the symbolic expression construction.

OP_XOR	OP_ADD	OP_LSL	OP_CONCAT
OP_AND	OP_MUL	OP_LSR	OP_EXTRACT
OP_OR	OP_POW	OP_ASR	OP_ZEXT
OP_NOT	OP_SUB	ARRAY	OP_SEXT

3. **Leak Set Domain**: $\mathcal{L}$, the set of vectors of all sizes whose components are sets of symbolic expressions $\mathcal{L} = \bigcup_{i \in \mathbb{N}^+} \mathbb{L}^i$, with $\mathbb{L} = \mathcal{P}(\mathcal{E})$.
4. **Stability Domain**: $\mathcal{S}$, the set of Boolean vectors of all sizes representing stability, $\mathcal{S} = \bigcup_{i \in \mathbb{N}^+} \mathbb{S}^i$, with $\mathbb{S} = \{0, 1\}$.

Definition 5 (Valuation of a wire)**.** The valuation of a wire w of width $|w|$ at cycle t is the tuple $\alpha_w^t = \langle {}^c\alpha_w^t, {}^e\alpha_w^t, {}^l\alpha_w^t, {}^s\alpha_w^t \rangle \in \mathbb{B}^{|w|} \times \mathbb{E}^{|w|} \times \mathbb{L}^{|w|} \times \mathbb{S}^{|w|}$, where ${}^c\alpha_w^t$ is the concrete value of the wire at cycle t, ${}^e\alpha_w^t$ its symbolic expression, ${}^l\alpha_w^t$ its *LeakSet* and ${}^s\alpha_w^t$ its stability.

In the following, we note $w[i]$ the i^{th} bit a wire w. We also note ${}^cf_g$, ${}^ef_g$, ${}^lf_g$ and ${}^sf_g$ the functionality of the gate g in each domain. For sake of clarity, only some gate functionalities in each domain are described (Tables 3, 4 and 5).

A simple circuit example is given in Figure 4 with valuations for each wire, in which symbolic expressions are simplified, e.g. $\texttt{OP\_AND}(\texttt{SYMB}(x), \texttt{CST}(0))$ becomes $\texttt{CST}(0)$.

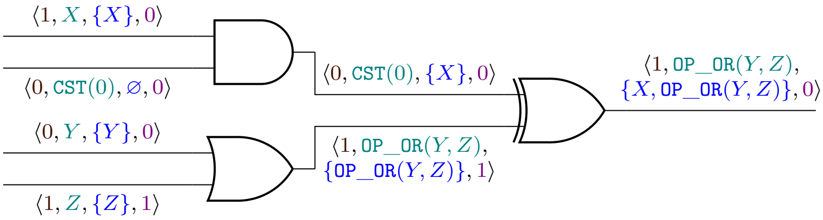


Figure 4: Circuit valuations with $X = \texttt{SYMB}(x)$, $Y = \texttt{SYMB}(y)$ and $Z = \texttt{SYMB}(z)$.

For a wire w at cycle t, the valuation α_w^t is computed with the application of an evaluation function for each domain, detailed in Sections 3.3.1, 3.3.2 and 3.3.3. Moreover, since I represents the set of input wires, a valuation is provided for each domain by the input sequence for every instant, following Equation 7.

$$\alpha_{i \in I}^t = \left\langle {}^c\alpha_i^t \in \mathbb{B}^{|i|}, {}^e\alpha_i^t = \begin{cases} \texttt{CST}({}^c\alpha_i^t) \\ \texttt{SYMB}(name) \end{cases}, {}^l\alpha_i^t[j] = \{{}^e\alpha_i^t[j]\} \forall j < |i|, {}^s\alpha_i^t = \vec{0} \right\rangle \tag{7}$$

3.3.1 Concrete and Symbolic Valuations

The evaluation functions of a wire w for the concrete (Equation 8) and symbolic (Equation 9) domains for a cycle t, named ${}^c eval(w, t)$ and ${}^e eval(w, t)$ respectively, are defined inductively until reaching inputs or registers along their cone of influence.

$$ {}^c eval(w, t) := \begin{cases} {}^c\alpha_w^t & \text{if } w \in I \\ t = 0 \ ? \ {}^c\alpha_w^0 : {}^c\alpha_{w_r^{in}}^{t-1} & \text{if } \exists r \in R \text{ s.t. } w = w_r^{out} \\ {}^cf_g(\dots, {}^c eval(w_i, t), \dots) & \text{else, } \exists g \text{ s.t. } w = w_g^{out}, \forall w_i \in W_g^{in} \end{cases} \tag{8}$$

For both domains, if w is the output wire of a register r, its valuation is the one of the input wire of r at the previous cycle. Otherwise, the valuation in both domains of a wire w output of a gate g, is the application of the functionality function (Table 3) ${}^cf_g$ or ${}^ef_g$ to all its inputs projected in the domain. For the symbolic domain, if w holds a symbolic constant $\texttt{CST(x)}$, $\texttt{x}$ must correspond to the concrete valuation ${}^c\alpha_w^t$.

$$ {}^e eval(w, t) := \begin{cases} {}^e\alpha_w^t & \text{if } w \in I \\ t = 0 \ ? \ {}^e\alpha_w^0 : {}^e\alpha_{w_r^{in}}^{t-1} & \text{if } \exists r \in R \text{ s.t. } w = w_r^{out} \\ {}^ef_g(\dots, {}^e eval(w_i, t), \dots) & \text{else, } \exists g \text{ s.t. } w = w_g^{out}, \forall w_i \in W_g^{in} \end{cases} \tag{9}$$

Table 3: AND, OR and XOR functionality computation in the $\mathbb{B}^n$ and $\mathbb{E}^n$ domains.

Domains	Inputs	Definition of ${}^c f_g$ and ${}^e f_g$
$\mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$	$({}^c\alpha_{w1}, {}^c\alpha_{w2})$	${}^c f_g(\dots) = {}^c\alpha_{w1}$ AND/OR/XOR ${}^c\alpha_{w2}$
$\mathbb{E}^n \times \mathbb{E}^n \to \mathbb{E}^n$	$({}^e\alpha_{w1}, {}^e\alpha_{w2})$	${}^e f_g(\dots) = \texttt{OP\_\{AND/OR/XOR\}}\ ({}^e\alpha_{w1}, {}^e\alpha_{w2})$

3.3.2 Stability Valuation

The evaluation function ${}^s eval(w, t)$ for a cycle t is bit-wise defined in Equation 10, for readability reasons. It has the same termination conditions as for the concrete and symbolic domains.

By default, no information is available on the origin of the inputs signals. Hence, no assumptions should be made about their stability: they are always defaulted to unstable.

When a wire is a register output, the stability of its bits is the bitwise syntactic equality of their current and previous symbolic expressions.

In most cases, the stability of the output of a combinatorial gate only depends on the stability of its inputs. However, particular input values for some gates allow to enforce stability using the symbolic valuation in conjunction with stability. For example, the stable symbolic input values CST(0) for AND gates and CST(1) for OR gates stabilise the output wire of these gates, regardless of the stability of the other input [MM24]. This requires that ${}^s f_g$ takes as inputs the valuation of input wires in both the symbolic and the stability domains. The AND and OR functionalities in Table 4 illustrate such a possible finer behaviour while the XOR and NOT functionalities only rely on their inputs stability valuations.

$$
{}^s eval(w,t)[i] := \begin{cases} {}^s\alpha_w^t[i] & \text{if } w \in I \\ t = 0\ ?\ {}^s\alpha_w^0[i] : {}^e\alpha_w^t[i] = {}^e\alpha_w^{t-1}[i] & \text{if } \exists r \in R \text{ s.t. } w = w_r^{out} \\ {}^s f_g(\dots, {}^s eval(w_j,t), \dots, & \text{else, } \exists g \text{ s.t. } w = w_g^{out}, \\ \qquad \dots, {}^e eval(w_j,t), \dots)[i] & \forall w_j \in W_g^{in} \end{cases} \tag{10}
$$

Table 4: Stability computation.

f_g	Domains	Inputs	Definition of ${}^s f_g[i]$
AND	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \to \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i])$ $\vee\ ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(0)))$ $\vee\ ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(0)))$
OR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \to \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	$({}^s\alpha_{w1}[i] \vee {}^s\alpha_{w2}[i])$ $\vee\ ({}^s\alpha_{w1}[i] \wedge ({}^e\alpha_{w1}[i] = \texttt{CST}(1)))$ $\vee\ ({}^s\alpha_{w2}[i] \wedge ({}^e\alpha_{w2}[i] = \texttt{CST}(1)))$
XOR	$\mathbb{S}^n \times \mathbb{S}^n \times \mathbb{E}^n \times \mathbb{E}^n \to \mathbb{S}^n$	$({}^s\alpha_{w1}, {}^s\alpha_{w2}, {}^e\alpha_{w1}, {}^e\alpha_{w2})$	${}^s\alpha_{w1}[i] \wedge {}^s\alpha_{w2}[i]$
NOT	$\mathbb{S}^n \times \mathbb{E}^n \to \mathbb{S}^n$	$({}^s\alpha_w, {}^e\alpha_w)$	${}^s\alpha_w[i]$

3.3.3 Leakset Valuation

The evaluation function ${}^l eval(w, t)$ is also bit-wise defined (Equation 11) for readability reasons. *LeakSets* are vectors of symbolic expressions sets. For an input wire, the initial valuation is a vector of size $|w|$ containing for each of its elements, the singleton composed of the bit of the symbolic expression of the same rank. If the symbolic valuation is a CST, the empty set ∅ is used instead, this case is referred to as *trivial* in the remainder of the paper.

For registers, as both the expressions of the output wire at the current and the previous cycles can leak, the *LeakSet* for cycle t is defined as the union of the sets of the same ranks. If a register output is stable, both expressions are identical, leading to a vector of singletons, analogously to the input *LeakSets* computation.

For a gate g, the *LeakSet* is computed on the inputs *LeakSets* as well as on its output stability and symbolic value. When the gate output is unstable, its valuation is a functionality-dependent combination of the inputs *LeakSets*. Whereas, when it is stable, its valuation is the vector of singletons containing the symbolic expression valuation bits (Table 5).

As for inputs, during the evaluation of a register output wire or a combinatorial gate *LeakSet*, sets only composed of symbolic constants are always replaced by empty sets.

$$ {}^{l}eval(w,t)[i] := \begin{cases} \{{}^{e}\alpha_w^t[i]\} & \text{if } w \in I \\ t = 0\ ?\ {}^{l}\alpha_w^0[i] : \{{}^{e}\alpha_w^t[i], {}^{e}\alpha_w^{t-1}[i]\} & \text{if } \exists r \in R \text{ s.t. } w = w_r^{out} \\ {}^{l}f_g(\dots, {}^{l}eval(w_j, t), \dots, & \text{else, } \exists g \text{ s.t. } w = w_g^{out}, \\ \qquad {}^{s}eval(w,t), {}^{e}eval(w,t))[i] & \forall w_j \in W_g^{in} \end{cases} \tag{11} $$

Table 5: AND, OR and XOR functionality computation in the $\mathbb{L}^n$ domain.

Domains	Inputs	Definition of ${}^{l}f_g[i]$
$\mathbb{L}^n \times \mathbb{L}^n \times \mathbb{S}^n$ $\times \mathbb{E}^n \to \mathbb{L}^n$	$({}^{l}\alpha_{w1}, {}^{l}\alpha_{w2},$ ${}^{s}\alpha_w, {}^{e}\alpha_w)$	${}^{l}f_g(\dots)[i] = \begin{cases} {}^{l}\alpha_{w1}[i] \cup {}^{l}\alpha_{w2}[i] & \text{if } \neg {}^{s}\alpha_w[i] \\ \{{}^{e}\alpha_w[i]\} & \text{otherwise} \end{cases}$

The stability evaluation requires the expression valuation, while the *LeakSets* evaluation requires both the expression and stability valuation. For efficiency reasons, the computation of the four valuations should be performed in a specific order. We present such an order in Algorithm 2, which performs a unique inductive traversal for a given wire at a given cycle, computing valuations in all the domains at once. By keeping in memory, during the cycle lifetime, all already computed valuations for internal wires, this traversal computes the state for the next cycle and the output for the current cycle.

3.4 Verification Management

The verification is done separately from the simulation of the model. A *verification manager* is responsible to extract simulated valuations in each domain and realise the appropriate verification in the given leakage model. The interface between the verification manager and the verifier is the function *verif* which takes as input a set of expressions in $\mathcal{E}$ and outputs whether they leak altogether by ensuring that the joint distribution of the expressions values is independent from the secret values it contains. The verifier is not aware of the leakage model; hence, the verification manager builds an expression set that is consistent with the considered leakage model by using the valuations computed through the mixed-domain simulation. We explain how the expression set is built for each leakage model in the next section; we then explain the optimisation regarding the number of wires which must be verified at each cycle, depending on the leakage model.

3.4.1 Expressions to Verify

Table 6 gives the expression set built for a wire w at a cycle t for each value of `g` and `t` in the `(g,t) 1-sw-probing` leakage model.

Algorithm 2: Mixed-domain simulation of the valuation of a wire w at cycle t.

```
Input   : A wire $w$, which is either a primary input or the output of a gate $g$
Output : The valuation $\alpha_w^t$ of the wire $w$ at cycle $t$
EVAL($w$,$t$):
    if $w \in I$ then return $\alpha_w^t$
    else if $g \in R$ then return $(t = 0 \; ? \; \alpha^0_{w_g^{out}} \; : \; \alpha^{t-1}_{w_g^{in}})$
    else
        /* With $w_{i1} \ldots w_{in}$, the wires in $W_g^{in}$                    */
        $\alpha^t_{w_{i1}}$ = EVAL($w_{i1}, t$)
        $\vdots$
        $\alpha^t_{w_{in}}$ = EVAL($w_{in}, t$)
        $cv = {}^c f_g({}^c\alpha^t_{w_{i1}}, \ldots, {}^c\alpha^t_{w_{in}})$                ▷ Concrete value
        $se = {}^e f_g({}^e\alpha^t_{w_{i1}}, \ldots, {}^e\alpha^t_{w_{in}})$             ▷ Symbolic expression
        $st = {}^s f_g({}^s\alpha^t_{w_{i1}}, \ldots, {}^s\alpha^t_{w_{in}}, {}^e\alpha^t_{w_{i1}}, \ldots, {}^e\alpha^t_{w_{in}})$       ▷ Stability
        $ls = {}^l f_g({}^l\alpha^t_{w_{i1}}, \ldots, {}^l\alpha^t_{w_{in}}, {}^e\alpha^t_w, {}^s\alpha^t_w)$                 ▷ LeakSet
        return $\langle cv, se, ls, st \rangle$
    end
```

Table 6: Expressions set built for a wire w at cycle t for each leakage model.

`(g,t) 1-sw-probing` model	Set of expressions to verify
`(g=0, t=0)`	$\{{}^e\alpha^t_w\}$
`(g=0, t=1)`	$\{{}^e\alpha^t_w, {}^e\alpha^{t-1}_w\}$
`(g=1, t=0)`	$\bigcup_{i=0}^{\lvert w\rvert-1} {}^l\alpha^t_w[i]$
`(g=1, t=1)`	$\{{}^e\alpha^{t-1}_w\} \cup (\bigcup_{i=0}^{\lvert w\rvert-1} {}^l\alpha^t_w[i])$

Value leakage. For the `(0,0) 1-sw-probing` leakage model, the singleton composed of the current expression of w, namely ${}^e\alpha^t_w$, is sent to the verifier.
Transition leakage. For the `(0,1) 1-sw-probing` model, $\{{}^e\alpha^t_w, {}^e\alpha^{t-1}_w\}$ the set containing the expressions from the current cycle and the previous one is sent to the verifier. This catches all possible combinations of both expressions, hence any possible transition.
Glitch leakage. For the `(1,0) 1-sw-probing` model, the sets of expressions in the *LeakSet* are flattened in a set that is sent to the verifier. This set captures all the glitches at the wire level.
Transition and glitch leakage. For the `(1,1) 1-sw-probing` model, the expression set sent is the singleton containing ${}^e\alpha^{t-1}_w$, the expression of w at the cycle $t-1$, with all the expressions of the flattened set of w at cycle t.

We can note that the verification of the expression built for the case with transitions or glitches encompasses the verification performed for the value leakage model. Similarly, the verification with both transitions and glitches encompasses the value leakage model, the transitions leakage model and the glitches leakage model.

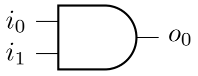
3.4.2 Wires to Verify

The straightforward verification strategy is to verify each wire at each cycle by sending to the verifier the expression set built according to the considered leakage model (cf. Table 6). In practice, many of the built expression sets are either empty or only composed of symbolic constants `CST(x)`, and thus are considered trivial and should not result in a verification request.

Table 7: Wires w to verify at cycle t for each leakage model.

`(g,t) 1-sw-probing model`	Wires to verify
(0,0)	$w \in W$ (all the wires)
(0,1)	$w \in W$ (all the wires)
(1,0)	$w \in W_r^{in}$ for $r \in R$, $w \in O$, $w \in W^{split}$, $w \in W_g^{in}$ with ${}^{s}\alpha^{t}_{w_g^{out}} \neq \vec{0}$, w other input of multiplexer whose selector is stable
(1,1)	$w \in W$ (all the wires)

We explain in this section the optimisations regarding the number of verification requests that can be applied by the verification management, depending on the considered leakage model. Table 7 gives for each leakage model the wires to verify at each cycle.



	${}^{e}\alpha_{i_0}$	${}^{l}\alpha_{i_0}$	${}^{s}\alpha_{i_0}$	${}^{e}\alpha_{i_1}$	${}^{l}\alpha_{i_1}$	${}^{s}\alpha_{i_1}$	${}^{e}\alpha_{o_0}$	${}^{l}\alpha_{o_0}$	${}^{s}\alpha_{o_0}$
$t-1$	$\texttt{CST}(0)$	$\varnothing$	0	$\texttt{k} \oplus \texttt{m}$	$\{\texttt{k} \oplus \texttt{m}\}$	0	$\texttt{CST}(0)$	$\{\texttt{k} \oplus \texttt{m}\}$	0
t	$\texttt{CST}(1)$	$\varnothing$	0	$\texttt{m}$	$\{\texttt{m}\}$	0	$\texttt{m}$	$\{\texttt{m}\}$	0

Figure 5: Example illustrating the need to verify each wire in the `(0,1)` and `(1,1)` `1-sw-probing` models. Input i_1 leaks `k` at cycle t while output o_0 does not.

Value and transition leakage. For the `(0,0)` and `(0,1)` `1-sw-probing` leakage models, the verified expression set contains the current (and the previous, for transitions) expression of the wire. We cannot reduce the number of verified wires, as an expression computed by a gate g can leak (e.g. $(k \oplus m) \oplus m$) while an expression of a gate h taking g as input does not $(m' \oplus ((k \oplus m) \oplus m))$. All the wires of the design must then be verified at each cycle. Figure 5 illustrates this requirement with an example in which the wire i_1 leaks when considering transitions whilst the result of the gate o_0 does not.

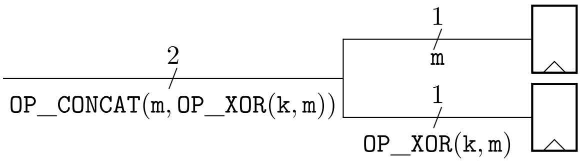


Figure 6: Example illustrating the need to verify wires which are split and partially used by a gate, with $\texttt{k} = \texttt{SYMB}(K)$ and $\texttt{m} = \texttt{SYMB}(M)$. The 2-bit wire leaks `k` while both 1-bit wires, each input of a register, do not.

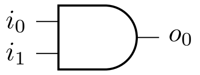
Glitch leakage. When only glitches are considered, not all wires need to be verified. For the `(1,0) 1-bit-probing` model without stability, *LeakSets* can only increase in size when traversing gates, and become maximal when reaching primary outputs of the design $(w \in O)$ or registers $(w \in W_r^{in}$ for $r \in R)$. However, when considering stability, this assumption does not hold anymore, because stability precisely allows the output *LeakSet* of a gate to be reduced. This is in particular the case for the inputs of gates whose outputs are stable (cf. Table 5) or the non-selected inputs of MULTIPLEXOR gates whose selectors are stable. Thus, these wires must be verified to account for the effect of stability. Finally, in the `(1,0) 1-sw-probing` model, *LeakSets* can also be reduced when a signal is split, as shown in Figure 6. Therefore, the set of wires that are split, which we denote W^{split}, must also be verified.

Glitch and transition leakage. For the `(1,1) 1-sw-probing` model, the built expressions include the expression of the wire at the previous cycle (cf. Table 6). As in the value leakage model, we must verify all the wires. The example given in Figure 5 illustrates why this is also required for this leakage model.

However, we can modify the expression set sent to the verifier (cf. Table 6) to the one containing all the elements of flattened sets of w at cycle t and $t-1$ i.e. the expression: $(\bigcup_{i=0}^{|w|-1} {}^{l}\alpha_w^{t-1}[i]) \cup (\bigcup_{i=0}^{|w|-1} {}^{l}\alpha_w^{t}[i])$. This expression set does not include the expression of the wire at $t-1$ but its *LeakSet* at $t-1$, which is correct because it over-approximates the leakage of the expression. This enables to reduce the number of verification requests: we do not need to verify all wires, but only those described in Table 8. In particular, the input wires of stable gates at cycle $t-1$ must be verified at cycle t. Figure 7 presents a circuit that would not reveal secret leakage when applying the over-approximation if we did not consider the stability at $t-1$.

Table 8: Wires w and associated expressions to verify at cycle t for the `(1,1) 1-sw-probing` model with over-approximation.

Expression to verify	$(\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t-1}[i]) \cup (\bigcup_{i=0}^{\lvert w\rvert-1} {}^{l}\alpha_w^{t}[i])$.
Wires to verify	$w \in W_r^{in}$ for $r \in R$, $w \in O$, $w \in W^{split}$, $w \in W_g^{in}$ with ${}^{s}\alpha_{w_g^{out}}^{t} \neq \langle 0 \rangle$ or ${}^{s}\alpha_{w_g^{out}}^{t-1} \neq \langle 0 \rangle$, w the non-selected inputs of a MUX with stable selector at cycle t or $t-1$



	${}^{e}\alpha_{i_0}$	${}^{l}\alpha_{i_0}$	${}^{s}\alpha_{i_0}$	${}^{e}\alpha_{i_1}$	${}^{l}\alpha_{i_1}$	${}^{s}\alpha_{i_1}$	${}^{e}\alpha_{o_0}$	${}^{l}\alpha_{o_0}$	${}^{s}\alpha_{o_0}$
$t-1$	CST(0)	$\varnothing$	1	k $\oplus$ m	{k $\oplus$ m}	0	CST(0)	$\varnothing$	1
t	CST(1)	$\varnothing$	0	m	{m}	0	m	{m}	0

Figure 7: Example showing the need to verify, at cycle t, the input wires of a stable gate at the cycle $t-1$. In the `(1,1) 1-sw-probing` model, input i_1 leaks `k` at cycle t while output o_0 does not.

Our approach considers `sw-probing` models, using the wires width specified in the HDL description of the design to analyse. However, one may want to consider 1-bit wires. In this case, using `(g,t) 1-bit-probing` models, all the reductions of the wires to verify presented above are still sound, but the verification of wires that are split is no more necessary. Verification in these more limited probing models does not enable the detection of violation of mask independence on a wire. Moreover, there are many more verification requests as all the bits of each wire to verify induce a verification request.

4 Implementation

The section describes aLEAKator, the framework that implements the mixed-simulation method proposed in Section 3. We first present the implementation of the verification flow, illustrated in the Figure 8, before explaining some of its key points.

4.1 Implementation Flow

Model generation. This first step is based on `yosys` [Wol13] and a modified version of its concrete simulation backend, `cxxrtl`.

`Yosys` is provided with HDL source files in Verilog, SystemVerilog, or VHDL (via GHDL). It then performs a partial synthesis—applying all synthesis steps except technology mapping—first on the CPU core (if present), and then on the complete system-on-chip or hardware accelerator.

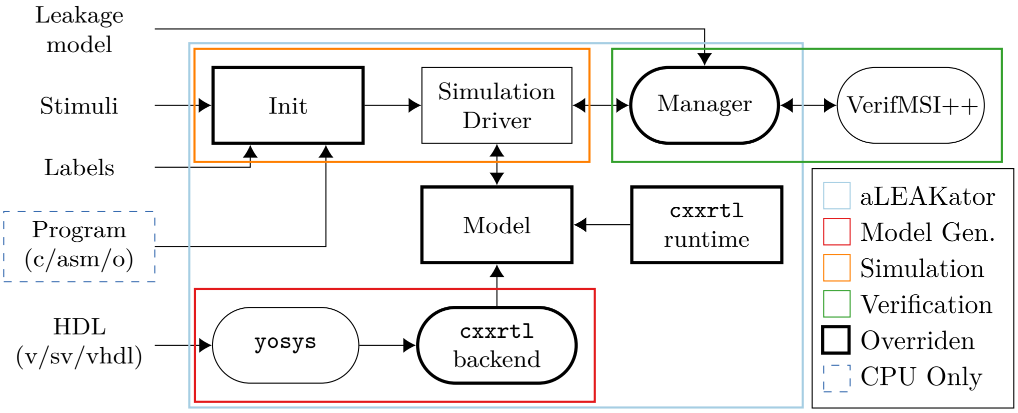


Figure 8: Implementation of aLEAKator.

The `cxxrtl` backend is subsequently invoked to generate a C++ model that can be simulated, which is compiled against a custom version of the `cxxrtl` runtime embedding all domain evaluation functions. This compilation is performed once for all executions of the hardware or programs on CPU. Said programs are compiled independently and loaded at simulation time.

The gates handled by aLEAKator are listed in Table 9. By operating at the level of high-level gates, we are able to verify circuits independently of the specific technological implementation of these gates, relying instead on the generic behavioural specifications, such as those provided as examples in Tables 3,4, and 5. While verifying a netlist mapped to a specific technology would also be feasible with aLEAKator, the tool is not currently designed for that purpose.

Table 9: Gates handled by aLEAKator.

Bitwise	Comparison	Arithmetic	Shifts	Resize	Miscellaneous
bit_not	ucmp	add	shl	(r)trunc	is_zero
bit_and	scmp	sub	shr	(r)zext	is_neg
bit_or	equal	neg	sshr	sext	mem_write
bit_xor	not_equal	mul		blit	mem_read
				repeat	register

Simulation. This second step is performed by feeding input stimuli to the model and triggering its evaluation. The CPU specific element presented in Figure 8 is the program. aLEAKator is able to handle any C, assembly or object program given by the user without recompiling the model by generating a plug-and-play memory loader including a bootloader for each program. Along with the program, the user provides the labels for the variables stored in memory or registers and can implement helper functions that are triggered at main function first and last instructions or at each cycle if needed. All programs are cross-compiled for the target architecture using the `llvm` compilation infrastructure, ensuring a seamless integration in the aLEAKator's toolchain. The simulation driver computes for all wires, the valuations in the four domains.

Verification. This third step is performed by the verification manager, which extracts from the model simulation the valuations in all domains for all wires. Then, thanks to its memory of the previous cycle valuations, it performs the needed verifications for the configured leakage model. Simulation and verification are performed in an interleaved manner until the end of the program. Depending on the verification result and the configuration, the simulation can also terminate after the first secret leakage. aLEAKator additionally provides the user with detailed information about the nature of the eventual secret leakages such as the simulation cycle number, their description (i.e., which symbolic expression, *LeakSets* or combination performed), their location (HDL source file, line number and wire name).

In order to build and manipulate its expressions, aLEAKator relies on an external verifier called `VerifMSI++`, which is an enhanced C++ version of `VerifMSI` [MT23], with bug fixes and more simplification rules. As a C++ library, it proves to be a good candidate for interfacing with aLEAKator. `VerifMSI++` allows the representation of complex symbolic expressions, their simplification, and implements an algorithm to determine if a given set of expressions statistically depends on the secret variables it contains, for a given security property.

4.2 Simulator Modifications and Verification Manager

Various adaptations of the concrete simulator `cxxrtl` in both the backend, for the model generation, and the runtime — now embedding the domain rules — have been done.

Model adaptation. As we want to verify the output of each individual gate, `cxxrtl` is configured to expose each wire of the netlist, i.e. it does not optimise parts of it by chaining gates in the simulation, but stores each gate output in the simulation state. For the same reasons, each implicit gates such as registers and multiplexers are made explicit in the generated model.

Overloaded memory accesses. Memory reads and writes are overloaded in such a way that, when specified, a user-defined function which takes as parameter the requested memory index wire, can be called. This enables the mixed-simulation of tabulated ciphers, which relies on a masking translation property on the symbolic memory reads (See `AES-Herbst` example in Section 5.2.2).

Restrictions on circuits. We restrict the circuits handled by aLEAKator to those that can be represented as Mealy machines. They must also be exempt of combinatorial feedback loops, so that there exists an evaluation scheduling which enables the evaluation of each gate once to fully determine the outputs and next state. We only consider circuits with a single clock domain and in which the synchronous gates are all sensitive to the same clock edge. Besides, `cxxrtl` has the following limitation: it considers that using a given bit of a signal to compute another bit of the same signal without crossing a register is a combinatorial feedback loop. For circuits exhibiting this behaviour, identifying these wires and splitting them is a required additional step to meet our assumptions. It is almost fully automated in our modified backend of `cxxrtl` but an additional pass may be needed for certain circuits. Wire splitting applied during the partial synthesis pass is done with a unique name generator that allows for the verification manager to recombine these wires for the support-aware verification. The verification manager is also made aware of the structure of the circuit via a file generated by `yosys` in addition to the model. This structural information is needed for the manager to detect which wires are a non-selected input of a multiplexer, as well as which wires are partially used (Figure 6).

Interface for extracting valuations. The verification manager, as described in Section 3.4.2, is in charge of extracting the valuations in the four domains from the model, for all internal wires. To this end, an interface allows the manager to query this information as well as to trigger a garbage collector for cleaning temporal *LeakSets*.

Expression cache. To enhance efficiency, a caching mechanism is implemented to verify identical expression sets only once. This optimisation significantly improves the implementation's performance.

Table 10: Verification results obtained by aLEAKator, Coco, Prover and Prolead.

Masked hardware	Verif/Total	aLEAKator		Coco		Prover	Prolead
	Cycles	Verdict	RAM (GB)	Verdict	RAM (GB)	Verdict	Verdict
SBox Present-TI NU	2/2	✗ <1s	0.01	✗ <1s	0.03	✗	✗
SBox Present-TI U	2/2	✓ <1s	0.01	✗* <1s	0.03	✓	✓
Present-TI NU	3/544	✗ <1s	0.01	-	-	-	✗
Present-TI U	3/544	✓ <1s	0.01	-	-	-	✓
AES-DOM	21/216	✓ 30s	12.3	✓ 26m30	3	-	-
Prince-TI	3/25	✗ 4.5s	2.2	✗ 4.6s	0.4	-	-
Prince-TI-Coco	3/25	✓ 5s	2.7	✓ 29s	0.4	-	-
Prince-TI-Coco	5/25	✗* 16s	8.0	✓ 3m45	1.7	-	-

5 Experimental Results

In order to prove the interest of the approach, we experimentally show on several benchmarks that aLEAKator obtains good results in terms of performance and accuracy.

We first present the verification results of masked hardware cipher accelerators, which highlights the concordance of the verification verdicts with the state of the art, and which are obtained in a shorter time. Second, we present the formal leakage analysis of masked software running on five different CPUs, which demonstrates the scalability and versatility of the approach, while verifying in particular proven leakage-free versions for all benchmarks. Finally, for two CPUs, we experimentally validate the results obtained with aLEAKator on two programs, by performing real power measurements of the same programs on the same targets. We show that when aLEAKator concludes that no secret leakage exists in the RR 1-sw-probing model, no such leakage can be observed in practice.

The experimental setup for all the experiments is an AMD Ryzen 9 7945HX computer with 92GB of RAM and 512GB of SSD swap memory, from the Dalek cluster [CAB25].

5.1 Masked Hardware Accelerators

Our analysis of already verified ciphers is performed on 3 main ciphers, from which we decline 8 benchmarks. We summarise the results obtained using aLEAKator on these benchmarks in Table 10. We also reproduce the verification on the same experimental setup using Coco which is, to the best of our knowledge, the only tool able to formally verify stateful ciphers in a leakage model considering transitions, glitches and stability. The hardware ciphers presented were all designed to be secure in bit probing models, thus, the verification performed is in the RR 1-bit-probing model.

Present is a lightweight block cipher primarily designed for hardware implementations, consisting of 31 rounds. Shahmirzadi *et al.* [SM21] proposed Present-TI, a TI-masked version of the Present cipher. Two variants of the cipher are available: one using a non-uniform (NU) SBox [EGMP17], and the other using a uniform (U) SBox [PMK$^+$11]. As shown by Prover [ZCF24], Coco experiment false-positives on both SBox versions while Prover can successfully verify that the uniform version is the only secure one. Prolead verified both versions as well as the full ciphers using them. aLEAKator corrects the eventual false-positives by using enumeration, allowing to confirm that the uniform SBox is secure and the non-uniform one is not. We also verify the three first cycles of the full cipher to obtain concordant results with Prolead but from cycle 3 onward, aLEAKator relies on enumeration. The fourth cycle is verified within 26 minutes with the same amount of memory usage. The results of Coco for the full cipher are not given here as it is already providing false-positive results for the SBox.

As Prover does not handle circuits in which inputs can vary over time, results are not given for Present, AES-DOM and Prince. Also, the version of AES-DOM and Prince verified by Prolead are not the same implementations and thus are not given either.

AES-DOM is a DOM-masked VHDL open source implementation of the AES-128 cipher proposed by Gross *et al.* [GMK16] with a configurable security order. It was previously verified, in its non perfectly interleaved version but with an eight stage SBox version, by Prolead to be (1,1) 1-probing secure (it is no longer available in the benchmarks of Prolead-V3 as of the date of submission). The perfectly interleaved version with seven stage SBoxes was verified by Coco up to the 21st cycle. We use aLEAKator to reproduce the results with the same configuration as Coco in Table 10. The accelerator is fed with 38 random bits per cycle. The shares of the key and plaintext are concatenated byte by byte and are provided to the circuit in 16 cycles. When the verification is pushed further, Coco times out while computing the 22nd cycle while aLEAKator can verify up to the 23th cycle. We note that on this benchmark, it is more than 50 times faster than Coco, with only 5.5 times more RAM usage.
Prince is a lightweight block cipher, smaller than Present, requiring only 11 rounds to compute. Božilov *et al.* [BKN22] proposed Prince-TI, a TI-masked implementation of the Prince cipher. It was proven unsecure by Coco due to a glitchy control signal, that aLEAKator identifies as well on the same multiplexer. A fixed version proposed by Coco's authors was verified up to cycle 7. aLEAKator is not able to conclude at cycle 5 and as enumeration would not be tractable, we interpret this result as a false-positive following Coco's result. Still, aLEAKator reproduces the secure verification verdict for the three first cycles in only the sixth of the time needed for Coco.

On summary, aLEAKator is able to reproduce known verification results, while still suffering from eventual false-positive results. On non trivial circuits, aLEAKator is up to 50 times faster than Coco, for a higher memory footprint.

5.2 Masked Software on CPUs

This section details the results obtained while verifying software ciphers on five CPUs. We first describe the target CPUs and benchmarks. We then present the complete verification results, and we end this section by showing, using power traces measured on real hardware, that using aLEAKator allowed to remove all observed real-world leakages.

5.2.1 Hardware Targets Configuration

To showcase the versatility of aLEAKator, we apply our method on five CPUs with memory subsystems, that we present in the following.
Ibex [4] is a CPU core implementing the open source 32-bit RISC-V ISA. Initially developed as part of the PULP platform, it is now maintained and developed by lowRISC. This core is highly configurable and implements a two or three stage pipeline (with or without write-back stage). It can also implement various multipliers and optionally a branch target ALU. The programs verified on the Ibex are verified in the three stage pipeline version. Experiments are made with the ibex simple system, an example bus and memory subsystem from the Ibex project.
Coco-Ibex [GHP+21][5] is a fork of the Ibex core secured by the authors of Coco. It features a read/write gated register file, a secure RAM, and gating mechanisms for the multiplier, shifter, adder, as well as control and status registers units. Each protection can be independently activated but we verify programs with all the protections enabled. Verification on this core uses the same bus as the Ibex core for accessing the unprotected ROM. We place our verification in the configuration of the authors of Coco [GHP+21], with the two-stage pipeline configuration without a write-back stage. Coco-Ibex implements a secure memory using fully mapped flip-flop cells. As a result, verifications for this CPU go deeper in the implementation of memories than other CPUs that use off-the-shelf SRAM

[4] Available at https://github.com/lowRISC/ibex.
[5] Available at https://github.com/isec-tugraz/coco-ibex.

cuts. We do not run programs such as the `AES-Herbst` which requires more than 512 bytes of RAM, since increasing RAM to 4096 bytes would double the Coco-Ibex core's gate count. The version of Coco-Ibex used in our experiments includes fixes to hardware bugs that prevented the use of certain common instructions, such as `c.jal`, which are required by our benchmarks. As Coco-Ibex implements countermeasures against side-channel attacks, some common leakage sources are mitigated by the hardware for this core. We note that following the authors' recommendations regarding the memory placement of shares is necessary to achieve non-leaking memory operations. Nevertheless, programs still need to be secured and verified, as the micro-architecture does not eliminate all secret leakages.

CV32E40P [GST+17][6] is a small and efficient open source 32-bit RISC-V processor with a 4-stage pipeline aimed at performance and energy efficiency. Originally maintained by the PULP platform; it is now maintained and developed by the OpenHW group. Experiments were made using the same memory subsystem as the Ibex.

Cortex-M3 and **Cortex-M4** are two general purpose 32-bit three stage pipeline ARMv7-M CPU designed for high-performance and low-cost platforms. Their HDL implementations are made available to us by Arm through the Arm Academic Access (AAA) program. An example AHB-Lite bus is provided with the cores, which we used for our verifications. We disabled tracing, debug and JTAG options for the experiments.

5.2.2 Verified Programs

For each program, two versions are provided, a non-secure and a secure one. Each secure version has been obtained by iteratively removing secret leakages using aLEAKator, until reaching a non-leaking result in the desired leakage model.

`DOM-AND` [GMK16] is an implementation of the AND function, masked with the `DOM` masking scheme. `DOM-AND-nsec`, the non-secure implementation, is an assembly version which has been proven secure when considering transitions at the GPR level. The `DOM-AND-sec-rr` implementations were designed to be secure in the `RR 1-sw-probing model`.

`Secmult` [RP10] is a masked implementation of the multiplication in GF(2^8). For the `Cortex-M3/M4` cores, the non-secure implementation, `Secmult-nsec`, is a `-O3` compiled version of the algorithm. For RISC-V processors, it refers to a manually-secured version in the value without glitches leakage model. The secure versions, named `Secmult-sec-rr`, refer to versions made to be secure in the `RR 1-sw-probing model`.

`AES-Herbst` [HOM06] is a masked tabulated implementation of the AES cipher using 6 masks, including the key schedule. The non-secure version, `AES-Herbst-nsec`, is a `-O3` compiled version of the cipher. As providing a secure version of the cipher in assembly would be a long task, the secure version, `AES-Herbst-sec-v`, is a `C`-level hardened implementation made to be secure in the `(0,0) 1-sw-probing model`. We still provide results of how aLEAKator performs when verifying this cipher in the `RR 1-sw-probing model` but we note that leaking implementations are usually longer and more costly to verify than non leaking ones, while it is usually desirable to stop at the first leakage found.

This implementation is tabulated using symbolic indexes. To tackle this issue, we used the overloading feature of the memory system for each CPU, so that the symbolic expression given as array index is used to determine the result of the memory access. In particular, the masking scheme of this cipher requires to compute a table `SBox'` which is initialised as `SBox'[i ⊕ m] = SBox[i] ⊕ m'` for all `i` values $\in$ [0; 255], with `m` and `m'` being two masks in the scheme. Upon this access, this expression transformation can only be done with this overloading feature, and this initialisation information.

[6] Available at https://github.com/openhwgroup/cv32e40p.

5.2.3 Software Verification Results

The verification results are presented in Table 11. For each CPU, we present each version of the three programs, except for `AES-Herbst` on the Coco-Ibex, as explained above.

As the verification manager excludes trivial verifications and cached results, the number of expressions sent to the verifier is far smaller than the number of wires of the circuit multiplied by the number of cycles. To better illustrate the effect of our over-approximation detailed in Section 3.4.2, we give two columns for the number of expression sets sent to the verifier. The first is *Expr. to Verify*, the number of expression sets that should be sent to the verifier when not using the over-approximation (while still using the cache and removing trivial expression sets). The second, *Verified Expr.*, is the number of expression sets sent to the verifier in practice, using the over-approximation. We emphasise that the difference between these two columns is the number of unique verifications which were not performed thanks to the over-approximation. This difference is only apparent for verifications in the `RR 1-sw-probing model` as all wires must be verified when considering the `(0,0) 1-sw-probing model`.

The *Leak. Cycles* column reports the number of cycles, among the cycles of the execution (column *Cycles*), for which at least one expression set cannot be proven non-leaking by the verifier.

We also report the time and RAM consumption for each program verification. We note that different CPUs can cause different RAM usage in simulation. This behaviour can be explained by the gating of some operators, like the multiplier or the shifter in the Ibex and CV32E40P for example. Non-gated operators tend to generate more symbolic expressions and, depending on the operator, expressions that are harder to verify. The RAM usage difference is also due to the number of gates, the number of simulated cycles and the number of already found leakages. On complex programs such as `AES-Herbst` RAM consumption is high but, as our experiments have been performed using swap, we consider this usage acceptable.

The various optimisations are an essential part of aLEAKator. When running `AES-Herbst-sec-v` on the Cortex-M3, over 80 millions trivial verifications are excluded, then around 2 million cache hits occur, allowing for a significant reduction of the verification time and RAM consumption. The over-approximation allows for the reduction by an average factor of 2.2 and up to a factor of 4.4 – for the CV32E40P's `Secmult` – of the number of unique expression sets to verify.

aLEAKator is efficient as it can verify ciphers requiring thousands of computation cycles, such as `AES-Herbst`, within the `RR 1-probing model` in 19 to 39 minutes, depending on the targeted CPU. When operating under a value leakage model, the verification can be completed in as little as five minutes on a Cortex-M3 processor.

The major results from the verification are that, for the `sec-rr` and `sec-v` versions of the program for which there is no leaking cycle, the implementations are proven secure on the corresponding targets in their respective leakage models. To the best of our knowledge, this is the first verification in both the `(0,0) 1-(sw/bit)-probing model` and the `RR 1-(sw/bit)-probing model` of a secure `AES-Herbst` in the `(0,0) 1-probing model` for the four processors. This paves the way for, if an assembly implementation was to be made, a secure implementation for those CPUs in the `RR 1-sw-probing model`.

5.2.4 Comparison with Real Power Measurements

We validate the results produced by aLEAKator by performing power measurements on real CPUs, on the programs `DOM-AND` and `Secmult`. For each program, both the `nsec` and the `sec-rr` versions have been considered. Measurements were made using

Table 11: Detailed aLEAKator verification results.

	SW	Ver.	F	Leak. Model	Cycles	Expr. to Verify	Verified Expr.	Leak. Cycles	Time	RAM (GB)
Cortex-M3	`DOM-AND`	`nsec`	9a	Value	48	630	630	0	1.8s	0.6
				RR	48	1 092	733	33	3.8s	1.0
		`sec-rr`	9b	RR	67	1 095	779	0	4.5s	1.0
	`Secmult`	`nsec`	11a	Value	209	2 916	2 916	53	5.1s	0.5
				RR	209	4 591	3 469	185	13.5s	1.4
		`sec-rr`	11b	RR	660	4 494	3 186	0	34.0s	1.7
	`AES-Herbst`	`nsec`		Value	6 924	74 707	74 707	3 050	3m46	37.1
		`sec-v`		Value	10 113	90 053	90 053	0	5m04	43.7
				RR	10 113	192 812	148 409	8 329	18m53	208.0
Cortex-M4	`DOM-AND`	`nsec`	10a	Value	48	649	649	0	1.9s	0.6
				RR	48	1 112	762	33	4.3s	1.1
		`sec-rr`	10b	RR	67	1 112	799	0	5.0s	1.0
	`Secmult`	`nsec`	12a	Value	203	2 998	2 998	36	5.7s	0.5
				RR	203	4 669	3 419	179	14s	1.4
		`sec-rr`	12b	RR	660	4 680	3 255	0	39s	1.8
	`AES-Herbst`	`nsec`		Value	6 924	77 227	77 227	3 050	4m13	38.0
		`sec-v`		Value	10 113	93 279	93 279	0	5m48	45.3
				RR	10 113	194 470	147 766	8 329	20m14	211.5
Ibex	`DOM-AND`	`nsec`		Value	37	705	705	0	5.1s	3.4
				RR	37	881	350	18	2.7s	1.0
		`sec-rr`		RR	58	833	355	0	2.1s	0.7
	`Secmult`	`nsec`		Value	638	12 703	12 703	0	11.4s	2.8
				RR	638	17 609	4 527	236	21.3s	4.5
		`sec-rr`		RR	646	17 270	4 427	0	20.5s	4.3
	`AES-Herbst`	`nsec`		Value	6 294	159 261	159 261	2 963	14m37	203.1
		`sec-v`		Value	8 869	185 508	185 508	0	21m41	226.8
				RR	8 869	230 463	90 050	6 651	22m08	301.0
CV32E40P	`DOM-AND`	`nsec`		Value	37	785	785	0	10.5s	9.9
				RR	37	860	299	17	5.8s	1.1
		`sec-rr`		RR	60	822	231	0	7.5s	0.8
	`Secmult`	`nsec`		Value	639	13 824	13 824	0	15.8s	3.4
				RR	639	12 461	2 886	234	1m14	5.0
		`sec-rr`		RR	651	12 228	2 783	0	1m15	4.8
	`AES-Herbst`	`nsec`		Value	6 294	176 150	176 150	2 762	21m38	267.6
		`sec-v`		Value	8 869	199 405	199 405	0	43m44	423.5
				RR	8 869	207 301	86 259	6 705	39m01	333.7
Coco-Ibex	`DOM-AND`	`nsec`		Value	51	537	537	0	1.5s	0.5
				RR	51	1 223	651	9	6.8s	3.5
		`sec-rr`		RR	61	1 194	961	0	7.6s	3.8
	`Secmult`	`nsec`		Value	724	2 033	2 033	0	34.3s	8.3
				RR	724	4 639	3 583	10	1m30	43.6
		`sec-rr`		RR	739	4 622	3 533	0	1m30	43.7
	SW	Ver.	F	Leak. Model	Cycles	Expr. to Verify	Verified Expr.	Leak. Cycles	Time	RAM (GB)

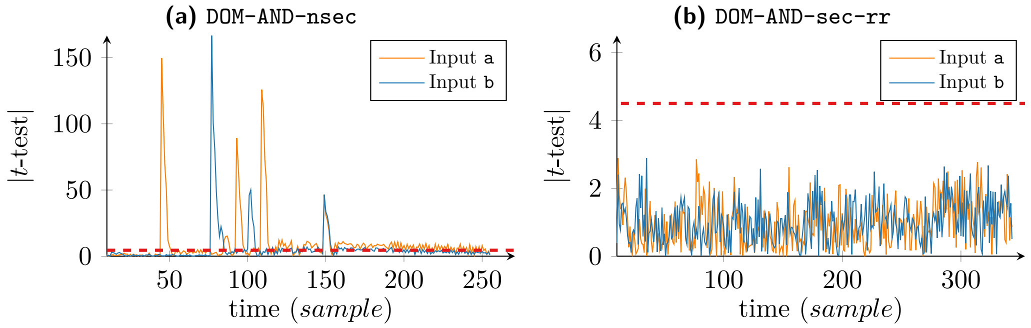


Figure 9: *t*-test values for the unsecure (a) and secure (b) DOM-AND on the Cortex-M3.

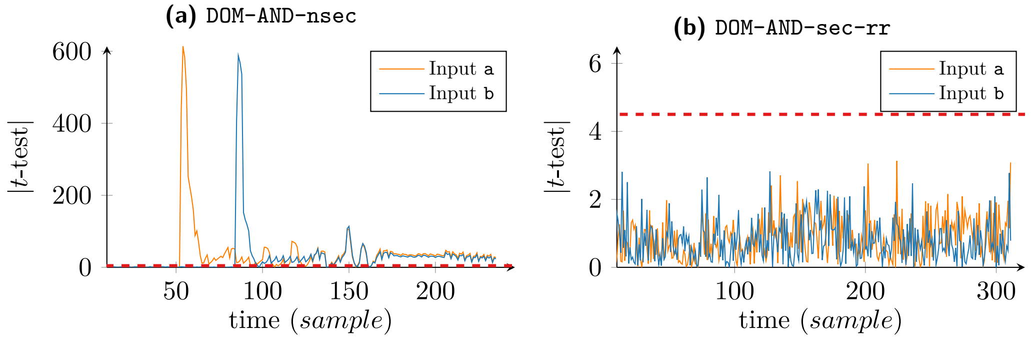


Figure 10: *t*-test values for the unsecure (a) and secure(b) DOM-AND on the Cortex-M4.

the ChipWhisperer CW1200 with the STM32F1 and STM32F3 target boards, embedding respectively a Cortex-M3 and a Cortex-M4 core [OC14].

Comparison of aLEAKator verdict with *t*-test measurements. The leakages reported by aLEAKator happen in many central parts of the CPU including the Load-Store Unit, the register bank, ALU ports and internal ALU signals. Figures 9, 10, 11 and 12 represent the evolution of the *t*-test value for each secret input for 500,000 traces, and for the entire duration of the program, with 4 samples per cycle. Each case is reported in Table 11 in the "F" column. The dotted red line represents the 4.5 value for the *t*-test, above which a secret is considered to be leaking. These figures show that all non secure versions exhibit very high leakage values, while all secure versions do not exhibit any. These results confirm that when aLEAKator reports no secret leakage, no such leakages are observed in practice.

5.2.5 In-depth analysis of leakage sources and cycle-accurate comparisons

We present three cycle-accurate comparisons between aLEAKator verdict and *t*-test values on the Cortex-M4 core: the DOM-AND-nsec, Secmult-nsec, and an additional Refresh micro-benchmark (Figures 13, 16 and 15 respectively). We additionally show how aLEAKator's output is used to precisely identify micro-architectural leakage sources.

aLEAKator verdict and *t*-test traces alignment. Four *t*-test values are provided per cycle by our experimental setup. As aLEAKator is cycle accurate, only the maximal value of this statistic for the cycle is plotted for the whole cycle. Moreover, as the whole physical system-on-chip may slightly differ from the Verilog model provided to us, we experienced some cycle shifts between the simulated and measured traces. Traces are resynchronised by stuttering items either in the simulation or measured traces. Once expanded, the number

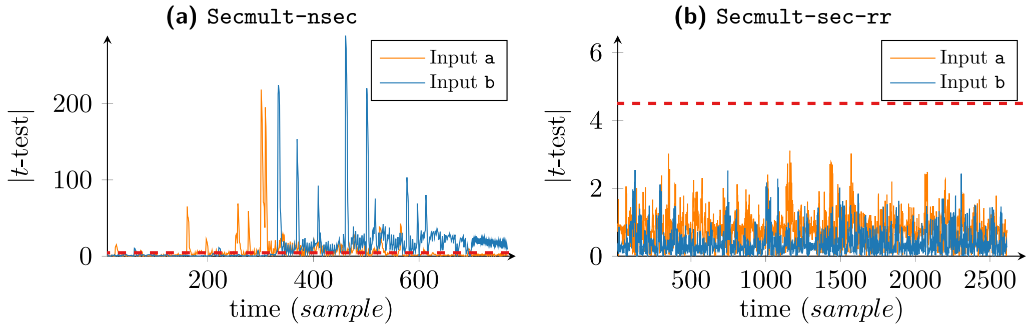


Figure 11: t-test values for the unsecure (a) and secure (b) `Secmult` on the Cortex-M3.

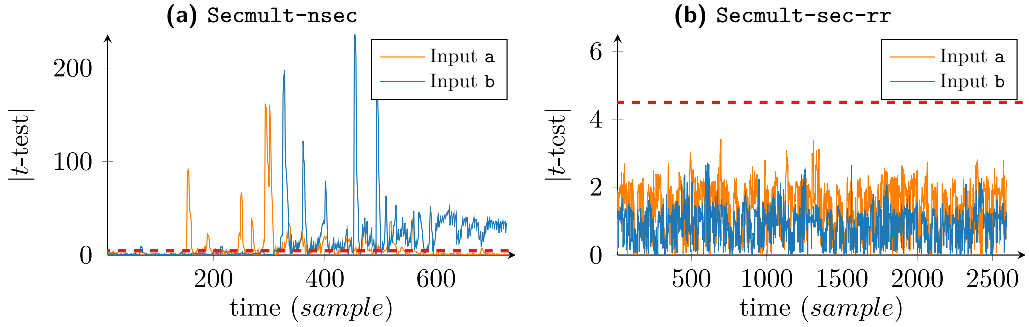


Figure 12: t-test values for the unsecure (a) and secure (b) `Secmult` on the Cortex-M4.

of cycles introduced by stuttering is less than 5% of the overall trace. Results show that all t-test values above the 4.5 limit are associated to a leakage detection by aLEAKator. We now illustrate for each use-cases some sources of leakage.

DOM-AND-nsec. For this use-case, each leakage identified with aLEAKator is observed using the t-test analysis (Figure 13). We focus on four different sources of leakage, which are numbered along the trace.

① The first leakage occurs at cycle 13. aLEAKator reports the flip-flop signal `HRDATAS_RAM` to be leaking in transition without glitches and points to the corresponding HDL description. A further analysis, using program counter and expression evolution on this signal reported by aLEAKator, enables to pinpoint the cause of the leakage in the assembly code given in Listing 1, whose purpose is to load both shares of the secret input `a` into the

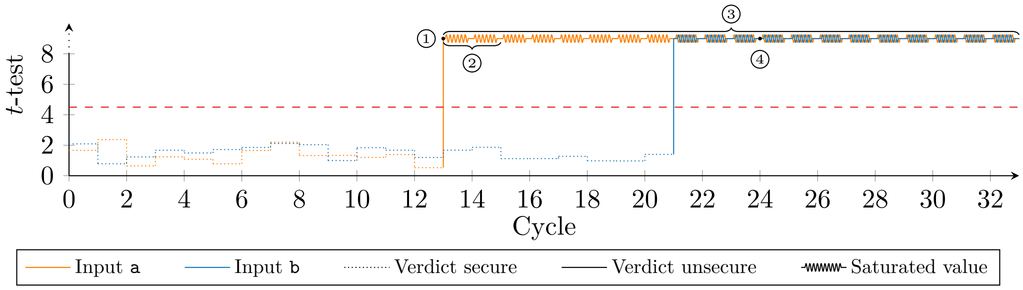


Figure 13: Cycle's max t-test value for each secret and corresponding aLEAKator `RR 1-sw-probing` verdict for the `DOM-AND-nsec` on the Cortex-M4.

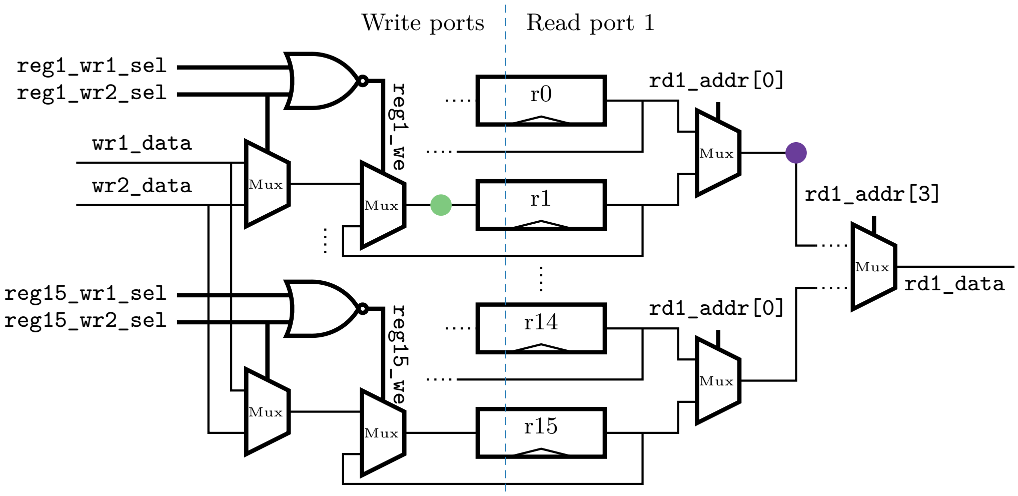


Figure 14: Simplified view of the register file of the Cortex-M4 core.

register file.

Listing 1: Load of shares a0 and a1 in DOM-AND-nsec

```
ldr r2, =a0   // PC-relative load @a0 in r2
ldr r0, [r2] // a0 in r0
ldr r2, =a1   // PC-relative load @a1 in r2
ldr r1, [r2] // a1 in r1
```

The first and third ldr are PC-relative loads, they read the address of the shares a0 and a1 from the code segment, transiting on the instruction bus from the FLASH memory. The second and fourth ldr read the data in RAM, transiting on the data bus, HRDATAS_RAM. The third ldr does not clear this bus, hence the transition between both shares is detected.

② The transition without glitches reported in ① at cycle 13 is also reported as a leakage in value with glitches (Register's case of Equation 11) and as a leakage in transition with glitches (line Expression to verify in Table 8) at both cycles 13 and 14.

③ aLEAKator reports a leakage by value with glitches from cycle 13 — performing the write of the second share of input a in the register file — onwards. This leakage is linked with the signal rd1_data of the u_cm4_dpu_regbank module, which is the output of the multiplexer tree corresponding to the read port 1 of the register file. An analysis of the *LeakSets* associated to this reported leakage shows that both shares are simultaneously present in the register file and may be recombined due to glitches. Part of read port 1 is pictured in Figure 14: if the multiplexer selector driven by the rd1_addr[0] wire is unstable, the multiplexer tree may glitch the r0 and r1 registers content. This explains why the *LeakSet* of the wire marked as ● contains both the *LeakSets* of r0 ({a0}) and r1 ({a1}), leading to a possible observation of the secret a.

④ At cycle 24, aLEAKator identifies a leakage related to the register file and the execution of the xor instruction eor r4, r4, r3 *//r4 = (a0 and b1), r3 = m*. The identified leakage source is the flip-flop holding r1 value in the register file. The design of the write path in the register file, as illustrated in Figure 14, may provoke glitches between the written data and the content of each register when its write enable signal is unstable. When writing the result of the eor instruction of DOM-AND-nsec, the reg1_we is unstable. As a consequence, the *LeakSet* of the wire marked as ● contains both the *LeakSets* of the write ports wr1_data and wr2_data as well as the *LeakSet* of the r1 register. The *LeakSet* of the write port wr1_data is equal to the *LeakSet* of the ALU output, i.e. $\{a0\&b1, m\}$.

The *LeakSet* of the `r1` register is $\{a1\}$. As a consequence the *LeakSet* of the wire marked as ● may leak `a`.

Similar phenomenons explain the leakages related to Input `b`, from the load of its shares at cycle 21 onwards.

Refresh micro-benchmark. aLEAKator can be used to identify leakages in seemingly non-leaking programs. Listing 2 details a program loading a share `a0` from a secret `a`, refreshing it with a mask `m` and storing it. Then, after clearing the register, the second share `a1` is loaded, refreshed with the same mask and stored.

The Armv7-M architecture used in the Cortex-M4 contains both 16-bit and 32-bit encoded instructions. Most instructions have multiple encodings while ensuring the same functional behaviour. For example, the `mov` instruction has three encoding, including two of 32-bit while the `ldr` instruction has two encodings, one 32-bit and one 16-bit. The micro-architectural behaviour of the instructions is dependent of the encoding [GHM22]. In the case of both the `mov` and `ldr` instructions, the written ALU ports differ with their encodings.

Since the LSU read and write data paths, along with the ALU and registers, are cleared between the loads, one could expect this implementation to be secure. However, as `r4` is cleared by `xor`ing it with itself, its value is written in both read ports of the ALU. The following `mov` is encoded as a 32-bit instruction which only clears the input port `A` of the ALU. After this, the 16-bit encoded `ldr` is executed and also only overwrites the input port `A` of the ALU. Finally, when the `xor` performing the refresh of `a1` is executed, input port `A` transitions from 0 to $a1$ and input port `B` transitions from $a0 \oplus m$ to m. Taken independently, none of these transitions can leak the secret `a`, but they cause a leaking transition in internal wires of the ALU (in the multi-cycles operators). This transition without glitches appears at cycle 18 in Figure 15. aLEAKator reports leakages in transitions with glitches in the cycles before and after, as the port `A` of the ALU can be recombined with the share `a1` that is newly loaded before the `xor`.

Listing 2: Refresh program

```
ldr r0, =a0
ldr r1, =a1
ldr r2, =m
ldr r3, =blk
ldr r4, [r0] // a0 in r4
ldr r5, [r2] // m in r5

eor r4, r4, r5 // Refresh a0
str r4, [r0]

// Clear write LSU path
str r3, [r3]
eor r4, r4, r4 // Clear r4
mov r4, 0x0 // Clear alu path?

// LSU Read path is cleared
ldr r6, [r1] // a1 in r12

eor r6, r6, r5 // Refresh a1
str r6, [r2]
mov r6, 0x0
```

These subtle leakages can be prevented by either clearing `r4` without rewriting it on the ALU ports with the `xor` (e.g `eor r4, r13, r13`), or using the 16-bit encoding of the `mov`, or using the 32-bit encoding of the `ldr`.

This real-world example is a key-argument of using automated tools such as aLEAKator: clearing data paths is difficult, it requires advanced knowledge of the data path taken by all instruction encodings and requires to take into account all the effects of surrounding instructions.

Secmult-nsec. Figure 16 demonstrates that all cycles for which a leakage is observed in practice are correctly identified as leaking by aLEAKator. Since this program is compiled with `-O3` optimisations, the masking in the source code is not perfectly preserved throughout the entire program. We also note that there are cycles without any observable leakages which are still reported as leaking by the tool, and we identified four main reasons for this discrepancy.

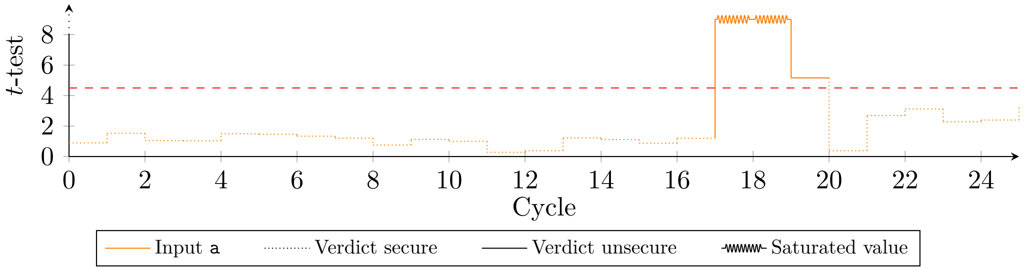


Figure 15: Cycle's max t-test value for each secret and corresponding aLEAKator `RR 1-sw-probing` verdict for the `Refresh` on the Cortex-M4.

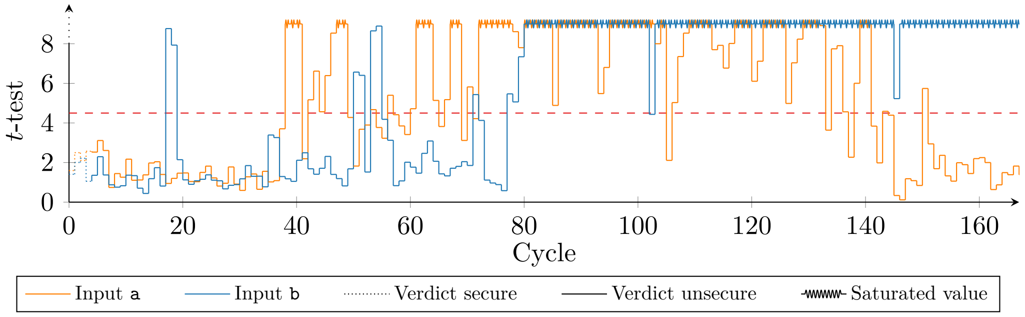


Figure 16: Cycle's max t-test value for each secret and corresponding aLEAKator `RR 1-sw-probing` model verdict for the `Secmult-nsec` on the Cortex-M4.

Firstly, an analysis of the compiled program shows that the four shares of `a` and `b` are stored within the same 32-bit memory word in the `.data` section. As a result, aLEAKator logically reports all memory reads of input shares as leaking in value without glitches (cycles 4-18 in Figure 16); even for byte loads, 32-bit words of data are put on the bus and later realigned and filtered in the LSU. Still, no leakage is observed in the t-test measurements, which we attribute to two reasons: 1) the t-test is performed at first order while having two shares in a single word is a second order leakage, and 2) from the point of view of a single secret variable, other shares in the same word behaves as a noise, which can be sufficient to hide the leakage.

Second, the analysis of the remaining detected but non-observable leakages shows that most are related to the presence of all shares in the register file, potentially leaking with glitches as illustrated in Figure 14. This may be linked to a third reason for the mismatch: the modelling of glitches is an over-approximation of real glitches, and only a subset of them may actually manifest in practice, depending on the synthesis timings.

Finally, the fourth reason for the mismatch between predicted and observed leakages is that we cannot guarantee that the verified RTL is exactly the same as the one used to produce the actual chip.

As a conclusion for this cycle-accurate analysis pointing to the source of leakages, we want to recall that all observed leakages are reported by aLEAKator. Thanks to the precise output of aLEAKator, all the reported leakages have been circumvented by introducing barriers, data variables reorganisation and shuffling instructions where needed, leading to the secure versions `DOM-AND-sec` and `Secmult-sec`.

Table 12: Higher-order verification results obtained by aLEAKator.

Masked hardware	Cycles	Security Order	Verification Order d	aLEAKator verdict	♯ d-uplets to Verify
`DOM-AND`	2/2	1	2	✗	240
`DOM-AND`	2/2	2	2	✓	666
`DOM-AND`	2/2	2	3	✗	15 540
`DOM-AND`	2/2	5	4	✓	450 662 52
`DOM-KECCAK`	2/2	1	1	✓	918
`DOM-KECCAK`	2/2	2	2	✓	1 047 552
`DOM-KECCAK`	2/2	2	3	✗	356 866 048
`DOM-KECCAK`	2/2	3	2	✓	22 028 942

5.3 Higher-order and Other Security Properties

Higher-order verification. Multiple propositions have been made regarding how to define higher-order verification for hardware and software. There can be spatial combinations, temporal combinations or a mix of both [HB21, MM22]. The common point of these propositions is that a verification at order d requires the analysis of d-uplets of expressions or *LeakSets*, which must be verified together. aLEAKator is able to handle both spatial and temporal combinations, as all the valuations required for such a verification are already computed at each cycle. However, no optimization are currently implemented, e.g. no optimisations regarding redundant probes. Therefore, aLEAKator is not yet meant to be used for higher-order verification on large circuits. Besides, even with optimisations, all existing higher-order verification methods suffer scalability issues because of the combinatorial increase in the number of d-uplets to verify. The latter grows as the number of combinations of d elements among p, with p being respectively the number of wires[7], simulated cycles, or the product of both, for spatial, temporal and mixed verification.

To support this discussion, we present in Table 12 the verdict of aLEAKator on the analysis of two hardware accelerators at different orders, reproducing existing spatial verifications on `DOM-KECCAK` [GSM17] (`Coco` and `Prover`) and `DOM-AND` [GMK16] (`VerifMSI`). The verdicts of aLEAKator are the expected ones. Moreover, results show the explosion in the number of d-uplets to enumerate and verify as the order grows; it quickly becomes intractable, as, for example, the verification at order 3 of a 4-share `DOM-KECCAK` would require the enumeration of more than 34 billions triplets.

Other security properties. aLEAKator main purpose is the verification of `d-probing` properties for software implementations. However, it can verify other security properties for hardware accelerators, such as NI and SNI [BBD+15, BBD+16], which are both currently implemented. As long as the needed information can be extracted from expressions or sets of expressions, there are no intrinsic limitation to support other security properties such as PINI [CS20].

In order to highlight this aspect, we show in Table 13 the results of NI and SNI verifications of common hardware gadgets `ISW-AND` [ISW03] and `DOM-AND` [GMK16] at order 2 with and without glitches. The `DOM-AND` for this experiment includes additional registers to account for the propagation on multiple cycles. These results corroborate with existing verifications from `VerifMSI` [MT23].

5.4 Discussion

When using aLEAKator on various circuits, we identified discussion-worthy elements, which we detail in this section.

[7] or bits for single bits leakage models

Table 13: NI and SNI results for hardware gadgets `DOM-AND` and `ISW-AND`

Masked hardware	Cycles	Security Order	Verification Order d	Property	With glitches	$\sharp$ d-uplets to Verify
`DOM-AND`	2/2	2	2	NI	✗	✓
`DOM-AND`	2/2	2	2	NI	✓	✓
`DOM-AND`	2/2	2	2	SNI	✗	✓
`DOM-AND`	2/2	2	2	SNI	✓	✗
`ISW-AND`	2/2	2	2	NI	✗	✓
`ISW-AND`	2/2	2	2	NI	✓	✗
`ISW-AND`	2/2	2	2	SNI	✗	✓
`ISW-AND`	2/2	2	2	SNI	✓	✗

Concretisations. Verified applications are assumed to have a control flow independent of input variables, represented as symbolic variables (secrets, masks and public variable). During the mixed-simulation, this translates into the constraint that all control wires should have a symbolic expression equivalent to a `CST` and their *LeakSets* should be $\varnothing$. However, this constraint may not hold in some processors, e.g. due to micro-architectural recombinations, for example when computing flags or bypasses. In such cases, aLEAKator propagates, on the control flow gates, non-constant symbolic expressions and non-empty *LeakSets*, expecting that they will be overridden before being stored in the CPU state. The worst case would be to obtain either a symbolic PC value or a symbolic instruction that would prevent from further simulating the circuit. In all our experiments, our assumption is sufficient and enables the simulation of complex programs on various CPUs.

Scalability. As for other formal approaches, not all circuits can be fully simulated. The data structure used for verification can cause memory issues when expressions are deep and involve many masks. The cost and simulability highly depend on the circuit structure and the generated expressions. This can be observed in Section 5 where some small hardware accelerators are heavier to simulate than full CPUs running programs.

False positives. Our method is not exempt of false positive leaking verdicts as we report an issue when the verification carried out by `VerifMSI++` finds a secret leakage or cannot conclude about the absence of secret leakage. Other formal tools suffer from this issue, as shown by `Prover` [ZCF24] with `Coco`.

No prior knowledge on CPU. aLEAKator was used on five CPUs without requiring prior knowledge on their implementation. The precision of the output of aLEAKator is valuable for understanding the leaks and designing leak-free programs.

Benefits of using expressions. Using symbolic expressions for both verification and hardware simulation offers many advantages. First, it allows the computation of stability by checking the equivalence of symbolic valuations across two consecutive cycles, without being limited to wires with constant values. Second, it enables the integration of custom rules for memory accesses, such as reading from a table with a symbolic offset. Third, it makes it possible to fall back to enumeration when verification cannot conclude for a small expression, helping to avoid false positives. Finally, the value of a signal can be made concrete at any time using the concrete input values available during simulation. This is useful for checking both the consistency of the expression and the symbolic state.

None of these advantages are possible with `Coco`, which only computes so-called correlation sets. These sets track correlations between the secret inputs and their combinations for each element in the circuit, but do not retain full symbolic information.

6 Conclusion

We presented a new method, implemented in a tool called aLEAKator, to formally verify cryptographic masked implementations running on CPUs as well as cryptographic hardware accelerators. This method relies on mixed-domain simulation, for which we formally defined the behaviour, promoting clarity in the verifications performed and enabling reproducibility of results. Additionally, both the implementation and the benchmarks are open source. While previous works were limited to formally verifying small programs such as S-boxes on a single CPU, we demonstrated the utility and efficiency of aLEAKator by verifying complete real-world software on multiple CPUs.

Furthermore, we showed—using actual power measurements on two CPU cores—that the verification results from aLEAKator align with the secret leakages observed in practice. We then presented a cycle accurate comparison between the output of aLEAKator and the *t*-test for multiple programs on the Cortex-M4 while providing precise and insightful leakage analyses. These verifications required only the HDL description of the circuits, with no additional knowledge about their implementation. Our method scales to real-world CPUs running widely-used cryptographic software, though this may require high RAM consumption. Nonetheless, proofs performed in the `RR 1-sw-probing` model can quickly guarantee the absence of first-order secret leakages.

References

[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 Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part I*, volume 11735 of *LNCS*, pages 300–318. Springer, Cham, September 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 Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Berlin, Heidelberg, April 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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

[BEOMHE19] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Side-channel robustness analysis of masked assembly codes using a symbolic approach. *Journal of Cryptographic Engineering*, 9:231–242, 2019.

[BKN22] Dusan Bozilov, Miroslav Knezevic, and Ventzislav Nikov. Optimized threshold implementations: securing cryptographic accelerators for low-energy and low-latency applications. *Journal of Cryptographic Engineering*, 12(1):15–51, April 2022.

[BWG$^+$22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In

Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 215–235, Cham, 2022. Springer International Publishing.

[CAB25] Adrien Cassagne, Noé Amiot, and Manuel Bouyer. Dalek: An unconventional and energy-aware heterogeneous cluster, 2025.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018*, volume 10815 of *LNCS*, pages 82–98. Springer, Cham, April 2018.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Berlin, Heidelberg, August 1999.

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

[EGMP17] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The first thorough side-channel hardware trojan. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 755–780. Springer, Cham, December 2017.

[EWS14] Hassan Eldib, Chao Wang, and Patrick Schaumont. Smt-based verification of software countermeasures against side-channel attacks. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 62–77, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[FGM$^+$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 TCHES*, 2018(3):89–120, 2018.

[GHM22] Arnaud de Grandmaison, Karine Heydemann, and Quentin L. Meunier. ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):3733–3744, 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 Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1469–1468. USENIX Association, August 2021.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, pages 3–3. ACM, 2016.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, Berlin, Heidelberg, August 1999.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 3–32. Springer, Cham, December 2021.

[GSM17] Hannes Gross, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of keccak. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 205–212, 2017.

[GST$^+$17] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[GXZ$^+$19] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. Quantitative verification of masked arithmetic programs against side-channel attacks. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 155–173, Cham, 2019. Springer International Publishing.

[HB21] Vedad Hadžić and Roderick Bloem. CocoAlma: A Versatile Masking Verifier. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*, volume 2 of *Conference Series: Formal Methods in Computer-Aided Design*, pages 14–23, Wien, 2021. TU Wien Academic Press.

[HHB24] Johannes Haring, Vedad Hadži´c, and Roderick Bloem. Closing the Gap: Leakage Contracts for Processors with Transitions and Glitches. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024:110–132, 2024.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *ACNS 06International Conference on Applied Cryptography and Network Security*, volume 3989 of *LNCS*, pages 239–252. Springer, Berlin, Heidelberg, June 2006.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Berlin, Heidelberg, August 1999.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Berlin, Heidelberg, August 1996.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 787–816. Springer, Cham, December 2020.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR TCHES*, 2022(4):311–348, 2022.

[MM24] Nicolai Müller and Amir Moradi. Robust but Relaxed Probing Model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024:451–482, 2024.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 01 2007.

[MPH23] Quentin L. Meunier, Etienne Pons, and Karine Heydemann. LeakageVerif: Efficient and Scalable Formal Verification of Leakage in Symbolic Expressions. *IEEE Transactions on Software Engineering*, 49(6):3359–3375, 2023.

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

[MT23] Quentin Meunier and Abdul Taleb. VerifMSI: Practical Verification of Hardware and Software Masking Schemes Implementations. In *Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT*, pages 520–527. INSTICC, SciTePress, 2023.

[MWO16] David McCann, Carolyn Whitnall, and Elisabeth Oswald. ELMO: Emulating leaks for the ARM Cortex-M0 without access to a side channel lab. Cryptology ePrint Archive, Report 2016/517, 2016.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Berlin, Heidelberg, December 2006.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *COSADE 2014*, volume 8622 of *LNCS*, pages 243–260. Springer, Cham, April 2014.

[PMK+11] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-channel resistant crypto for less than 2,300 GE. *Journal of Cryptology*, 24(2):322–345, April 2011.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Berlin, Heidelberg, August 2010.

[SCS+21] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 685–699. ACM Press, November 2021.

[SM21] Aein Rezaei Shahmirzadi and Amir Moradi. Second-order SCA security with almost no fresh randomness. *IACR TCHES*, 2021(3):708–755, 2021.

[SSB+21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS 2021*. The Internet Society, February 2021.

[Wol13] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys/, 2013.

[ZCF24] Feng Zhou, Hua Chen, and Limin Fan. Prover - Toward More Efficient Formal Verification of Masking in Probing Model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):552–585, Dec. 2024.

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. PROLEAD_SW probing-based software leakage detection for ARM binaries. *IACR TCHES*, 2023(3):391–421, 2023.