Formal Methods in Computer-Aided Design 2021

CocoAlma: A Versatile Masking Verifier

Vedad Hadžić
Graz University of Technology

Roderick Bloem
Graz University of Technology

***Abstract*—Masking techniques are an effective countermeasure against power side-channel attacks. Unfortunately, correctly masking a hardware circuit is difficult, and mistakes may lead to functionally correct circuits with insufficient protection. We present CocoAlma, a tool that formally verifies the side-channel resistance of stateful hardware circuits. Although CocoAlma was initially used to verify programs running on CPUs, we extended it to verify the security of several industrial masked hardware implementations. We give an overview of the tool's structure, implementation details, optimizations that make it faster and more scalable than its predecessor Rebecca, and changes that enable verifying the probing security of any stateful hardware circuit. Finally, we evaluate CocoAlma with masked implementations of the Prince and AES ciphers.**

***Index Terms*—Side-channels, Hardware masking, Formal verification**

I. Introduction

Integrated circuits that process sensitive data are susceptible to passive *side-channel attacks* like differential *power analysis*. Naturally, attackers are interested in the secret keys of symmetric ciphers because that would break the confidentiality of the processed data [22], [23], [26], [21]. Classical power analysis attacks exploit the correlation of the circuit's power consumption to bits of the secret key. Ultimately, the key is reconstructed using statistic analysis techniques in a series of key guesses [22], [27].

Masking is an algorithmic countermeasure against power analysis attacks. It relies on splitting all secrets and intermediate computations into multiple signals. The circuit is rewritten so that attackers can only reconstruct the original value if they can observe all the shares simultaneously. Masking techniques achieve this by introducing randomness into the circuit and destroying the correlation between the power-trace and the original data. Several masking schemes describe how to make circuits secure against side-channel attacks. Among them, *domain-oriented masking* [15] and *threshold implementations* [9] are well studied and widely adopted. The security of masked hardware circuits is expressed using the *hardware probing model* [2], [18], [4], where an attacker can read the values of d wires. Traditionally, engineers validate masked hardware implementations empirically by creating power traces and computing the correlations over many executions. Recently, however, we see several formal masking verification methods that can substantially reduce the costs of validating power side-channel resistance of software and hardware [2], [1], [11].

This work was supported by the *Austrian Research Promotion Agency* (FFG) through the Fermion project (grant number 867542).

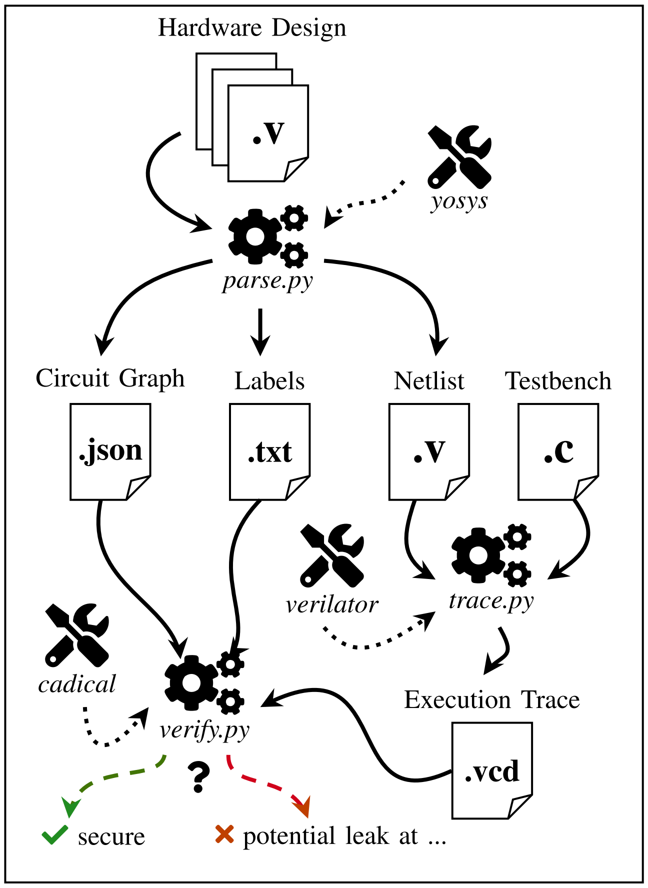

Figure 1. The workflow of CocoAlma showing the *parsing*, *tracing*, and *verification* phases, as well as their artifacts. At the end of the verification phase, CocoAlma either acknowledges that the analyzed design is secure or shows that a secret is leaked at a given location in the circuit.

CocoAlma is an open-source masking verifier[1] that assisted the hardening of a RISC-V processor[2] so it could safely execute masked software [13]. It considers the exact description of the hardware that runs the software and accounts for hardware leakage effects such as glitches. Figure 1 shows the workflow of CocoAlma. Starting with a hardware design written in Verilog, CocoAlma uses Yosys [31] to synthesize a flat gate-level Verilog netlist. Additionally, the parsing phase extracts a circuit graph of the synthesized design and creates a labeling template where the user can specify the contents of each register and input port of the circuit after the reset.

[1]https://github.com/IAIK/coco-alma
[2]https://github.com/IAIK/coco-ibex

https://doi.org/

COCOALMA uses a testbench provided by the user to simulate the netlist with Verilator [28], resulting in a *value change dump* showing how the internal signals changed throughout the execution. For the analysis of software running on RISC-V processors, COCOALMA additionally requires the RISC-V toolchain to compile programs and add them to the testbench before starting the simulation. The resulting execution trace is used to determine the value and glitching properties of each wire in the design. Afterward, the time-constrained probing model, initial state, simulation trace, and glitching information are encoded as a SAT problem and solved with CaDiCaL [3]. If the problem is unsatisfiable, no possible observation would leak any of the secrets. Otherwise, COCOALMA gives a precise description of leakage location, the secret bits that are leaked, and a variety of other debugging information.

Although COCOALMA was first used for analyzing software running on CPUs [13], its roots in the older verification tool REBECCA [4] can be leveraged towards stateful hardware verification of masked cipher implementations. Luckily, all the principles used in COCOALMA also apply to hardware masking verification with minor tweaks. In this paper, we document the inner workings of COCOALMA, its features, and show the extensions necessary for applying it to cryptographic accelerator modules. We present the following details about COCOALMA's implementation:

- In Section II, we define the supported probing models, emphasizing the newly supported *hardware probing model*, which allows us to prove the security of stateful hardware circuits. We also discuss the support for *random number generators*.
- In Section III-A, we give a breakdown of the *correlation set* methodology and show its encoding into a SAT formula in Section III-B. Here we give a precise description of the encoding, which is missing in the original publication [13], and more efficient than the encoding used in REBECCA [4]. Finally, in Section III-C, we describe details of several optimizations that reduce the size of the encoding and the number of probing locations. Here, the *hardware probing model* requires special considerations.
- In Section IV, we motivate and describe the execution-dependent correlation set simplifications. Additionally, we present the *stable signal detection* algorithm computing the *stability* of each control signal in Section IV-A. This optimization allows us to simplify the correlation sets even in the presence of glitches.
- In Section V, we demonstrate COCOALMA's capabilities by verifying the probing security of state-of-the-art masked implementations of the PRINCE [6], [12], [20] and AES [30], [7], [17], [15] ciphers as they are popular in the semiconductor industry. Additionally, we go over the debugging tools provided with COCOALMA, which allow a designer to locate the source of the leakage and see how leakage propagates through the circuit.

II. SECURITY MODELS

Masked implementations split all intermediate data signals x into $d+1$ uniformly random pieces x_i, with $x = x_0 \oplus \ldots \oplus x_d$. In practice, for $i \neq d$, the signal shares x_i are sampled from a random number generator, whereas x_d is chosen as $x \oplus x_0 \oplus \ldots \oplus x_{d-1}$ to fit the equality. This countermeasure tries to prevent an attacker, who can observe intermediate computations through side-channels, from learning anything about the processed data. When investigating whether a masked implementation is actually side-channel resistant, several security models describe the capabilities of an attacker and the real-world effects they can observe. COCOALMA implements three different probing models that consider different attacker capabilities and system behavior. More specifically, this work extends COCOALMA to support continuous probing as part of the *hardware probing model*.

Software probing model. The original probing model defined by Ishai et al. [18] considers the stable state of computations, ignoring hardware side-effects such as glitches and transitions. Their seminal paper says that an attacker in this probing model can choose d intermediate values that they can observe. The attacker can then interactively query the execution of the system several times with different inputs and starting states. The inputs of the computation are declared either (a) *public*, which means that learning them does not benefit the attacker, (b) fixed uniformly random values called *masks*, or (c) parts of a secret called *shares*. The attacker's goal is to learn all the shares of a secret and use them to reconstruct the secret value they are not supposed to know. Proving that an implementation is d-probing secure requires showing that no attacker adhering to this probing model can learn the secrets, irrespective of their strategy.

Time-constrained probing model.[3] When COCOALMA was first presented [13], its primary goal was verifying the masking of software programs running on an accurate description of the underlying hardware. Naturally, this required an adequate probing model that translates software probing into the hardware domain. The *time-constrained probing model* uses the gate-level description of the hardware and an execution trace generated by simulating the hardware running the software, instead of a purely algorithmic description. The goals of the attacker are the same as in the *software probing model*. However, this model is more realistic, as the attacker can probe d observation tuples (g, t), where g is a logic gate or register and t is a cycle in the execution trace. This gives an attacker access to all the intermediate values of gate g in cycle t, including all the values caused by hardware effects such as glitches and register transition leakage. The two parameters g and t are not coupled, meaning that the attacker can also probe the same gate in multiple clock cycles or even probe d different gates in the same clock cycle. Although this model limits each probe to observing only one clock cycle, instead of running throughout the computation, its inclusion of hardware effects significantly enhances the capabilities of an attacker.

[3]Barthe et al. [2] and Moos et al. [24] call this the *robust probing model*.

Due to the different signal timings in hardware, an attacker observing gate $g = a \odot b$ in this model would also observe the signals a and b in addition to g. Registers are synchronous elements triggered by a clock, making them the only hardware elements exempt from this phenomenon. Another effect that increases the attacker's capabilities is transition leakage, which causes the power consumption to correlate with the linear combination $g^{t-1} \oplus g^t$ of the old signal value in cycle $t-1$ and the new signal value in cycle t. Transition leakage applies to all hardware elements equally, including registers.

Hardware probing model. This paper extends the tool CocoAlma with a model where probes are not bound to one clock cycle like in the *time-constrained probing model*. The attacker's goals remain the same as before, only that in this more rigorous model, the probes record continuously throughout the whole computation. More precisely, instead of choosing a clock cycle for each observed location, the attacker observes all values, including those caused by glitches and transitions, that pass through a wire. In a sense, this is a more powerful rephrasing of the original probing model of Ishai et al. [18], as they also did not limit the duration of the probes for stateful circuits. As this model significantly increases the capabilities of an attacker, hardware designers employ random number generators to create fresh uniformly random masks in each clock cycle, intending to break any correlations that might otherwise be observed. These mask-generating circuits are usually not part of the masked hardware designs and are only used as black-boxes that provide random inputs to the masked circuit. We incorporate this in CocoAlma, allowing designers to label input ports of a circuit as *random*. The values read from these ports behave similarly to fixed *masks*, only that they represent a new mask in each clock cycle, which is then considered during verification. The semantics of *public* and *share* signals remains the same, and we even allow fixed *masks*, just like in the other probing models.

III. Verification Method

CocoAlma tries to verify the side-channel resistance of a masked implementation in one of the given security models. A correctly masked implementation computes the values of arbitrary logic functions without exposing the value of the secret to an attacker through intermediate computations. Therefore, a masked implementation must ensure that intermediate signals do not correlate with *secrets*; that is, the value of an intermediate signal should be statistically independent of all secrets. CocoAlma checks whether these properties hold by tracking the correlations of each logic operation throughout the computation [4], [13]. For instance, if a circuit were to compute the expression $f = a \wedge b$, then f correlates positively with a, b, and the constant $\bot$ because they have the same value in three out of four cases. For the same reason, f correlates negatively with the linear combination $a \oplus b$ because they only have the same value in one of four cases, *i.e.*, when both a and b are $\bot$. An exact algorithm that computes these correlations would solve the #SAT problem [14], meaning that computing correlations is at least #P-Complete [29], which is harder than NP by definition. Because of the structure of secrets and the uniform randomness of secret shares and masks, it is sufficient to track the correlations to linear combinations of the inputs [4]. Furthermore, the correlations yield a sound over-approximation that reduces the complexity of the problem and is also used in CocoAlma. In the following sections, we describe this over-approximation and its implementation, but refer to the soundness proofs in the original publication [4].

Table I
Propagation rules for stable and transient correlation sets

Gate type of f		Stable set S_f^t	Transient set T_f^t
Constant	$\bot$ or $\top$	$\{\bot\}$	$\{\bot\}$
Input Port	p^t	$\{p^t\}$	$\{p^t\}$
Negation	$\neg a$	S_a^t	T_a^t
Register	$\Leftarrow_R a$	S_a^{t-1}	S_a^{t-1}
Linear	$a \oplus b$	$S_a^t \otimes S_b^t$	$\langle T_a^t \rangle \otimes \langle T_b^t \rangle$
Non-linear	$a \wedge b$ $a \vee b$	$\langle S_a^t \rangle \otimes \langle S_b^t \rangle$	$\langle T_a^t \rangle \otimes \langle T_b^t \rangle$
Multiplexer	$c \, ? \, a \, : \, b$	$\langle S_c^t \rangle \otimes (S_a^t \cup S_b^t)$	$\langle T_c^t \rangle \otimes \langle T_a^t \rangle \otimes \langle T_b^t \rangle$

A. Correlation Sets

Instead of painstakingly computing the exact correlation factor for each linear combination of inputs, CocoAlma over-approximates the correlations. In particular, CocoAlma only considers whether the correlation factor is non-zero, and ignores its exact value. All linear combinations a gate correlates to are grouped together and tracked as so-called *correlation sets*. The exact correlations are approximated using propagation rules that determine the correlation set of $f = a \odot b$ by considering the correlation sets of a and b, as well as the used logic operation $\odot$. Using the previous example $f = a \wedge b$, we have shown that the correlation set contains all linear combinations of a and b, *i.e.*, $\{\bot, a, b, a \oplus b\}$. In contrast, $f = a \oplus b$ only correlates with itself, *i.e.*, the set $\{a \oplus b\}$, because the value of $a \oplus b$ coincides with $\bot$, a, and b in exactly half of the cases, yielding a correlation factor of zero. Consequently, knowing f would not reveal any information about a and b. In general, we cannot compute the correlation set of the output of a logical operation precisely from the correlation sets of its inputs, so CocoAlma over-approximates these sets.

Table I presents the propagation rules CocoAlma uses to compute the correlation sets of a gate using its inputs. The propagation rules define two kinds of correlation sets necessary for the verification: (a) *stable* sets S_f^t that define the normal behavior of a gate f, and (b) *transient* sets T_f^t that define the behavior of f in the presence of glitches and transition leakage effects. Both types of correlation sets are defined for each clock cycle t, as gates change their value over time. Although the hardware probing model only talks about these transient correlation sets, the stable correlation sets are necessary for synchronizing elements such as registers. For simpler exposition and encoding, Table I shows the computation of correlation sets using the operators $\otimes$ and

$\langle\cdot\rangle$. Here, $\otimes$ is the element-wise exclusive-or between two correlation sets, *i.e.*, $X \otimes Y = \{x \oplus y \mid x \in X, y \in Y\}$. The operator $\langle\cdot\rangle$ adds a correlation with $\perp$ to a correlation set, *i.e.*, $\langle X\rangle = X \cup \{\perp\}$.

The presented propagation rules are based on COCOALMA's original publication [13], [4] but were adapted for stateful hardware verification with continuously recording probes. Naturally, constants only correlate to $\perp$, and negations only change the sign of the correlation but do not impact the correlations themselves. As discussed previously, linear gates only correlate to the linear combination of the inputs, so the correlation set is computed as the element-wise exclusive-or of the inputs' correlation sets. For non-linear gates, the correlation set is computed similarly, only that in this case, a bias is introduced in each input's correlation set. Using the introduced notation, the correlation set of gate $f = a \wedge b$, where a and b are inputs, is computed as

$$\langle\{a\}\rangle \otimes \langle\{b\}\rangle = \{\perp, a\} \otimes \{\perp, b\} = \{\perp, a, b, a \oplus b\} \ . \quad (1)$$

For transient correlations, linear gates behave like non-linear gates. Glitches induced by different signal timings can force a gate to forward a constant or either of the inputs, in addition to the correct correlations. A multiplexer correlates to both of its data inputs a and b, as well as their linear combinations with the selector c, *i.e.*, $a \oplus c$ and $b \oplus c$. For the transient correlation set, COCOALMA assumes that all three input signals can be combined non-linearly.

When verifying masked software running on a processor, the input pins of the hardware design are not relevant, as they are part of the micro-architecture and not visible to the programmer. Secret shares, masks, and public values are all stored in both the RAM and the ROM, and for the verification process, we label their locations and simulate the design to execute a program [13]. Verifying masked hardware is different, as there are no such memory blocks, and the registers get cleared with a reset signal. Computation-relevant data, such as plaintexts, keys, and masks, is provided by the environment through the input ports of the circuit. Therefore we extend COCOALMA with support for input ports and introduce an appropriate propagation rule, which states that an input port only correlates to its value in cycle t. In our implementation, *public* values, *shares*, and *masks* have the same value throughout the execution of the circuit. However, input ports labeled as *random* are provided by an external *random number generator* and change their value in each cycle, and therefore, the correlation set also changes each cycle. In addition, to the support for input ports, we also optimized the propagation rules for registers. Since the probes in the *hardware probing model* record data continuously, we do not need to account for transition leakage because all values passing through a wire are recorded anyway.

Computing correlation sets from other correlation sets can result in over-approximations that include non-existent correlations. For example, representing the exclusive-or function $f = a \oplus b$ as $f = (a \wedge \neg b) \vee (\neg a \wedge b)$ would result in the spurious correlation set $\{\perp, a, b, a \oplus b\}$, when in reality f only correlates with $\{a \oplus b\}$. This means that a hardware designer applying this over-approximative method must be aware of false leakage reports and debug them properly. Oftentimes, as illustrated in this toy example, the over-approximative error can be fixed by either re-writing the circuit or removing the problematic correlation term from the correlation set.

However, despite being imprecise, this over-approximation is easy to encode and retains some useful information. For example, function $f = (a \oplus b) \wedge c$ is correctly claimed to correlate with $\{\perp, c, a \oplus b, a \oplus b \oplus c\}$, even though the correlation set of f was computed using the correlation sets of $g = a \oplus b$ and c. This result reflects the intuition that we cannot "remove" masking from a signal by combining it with another value, *i.e.*, the correlation set does not contain values where a appears without b.

B. SAT Encoding

The upper bound for the size of the correlation sets is exponential in the number of inputs, so COCOALMA cannot store or enumerate them explicitly and instead relies on an implicit encoding method that utilizes a SAT solver. While the used encoding is similar to the one presented by Bloem et al. [4], it was significantly optimized and streamlined in COCOALMA to simplify the implementation of all the propagation rules in Table I. As mentioned previously, the user needs to label each input port $p \in \mathcal{I}$ as either a *share* $s \in \mathcal{K}^i$ of the i-th secret, a fixed random *mask* $m \in \mathcal{M}$, a *random port* with a new value $r \in \mathcal{R}^t$ in each clock cycle t, or a public value that is ignored. For simpler notation, we do not implicitly associate correlation sets or propositional variables with clock cycles or gates in the circuit, and instead specify them with $\mathcal{C}_{-}$ and $\mathcal{P}_{-}$, where the subscript is used to differentiate them. In our SAT encoding, a correlation set $\mathcal{C}_x$ is represented by a set of propositional variables $\mathcal{P}_x = \{x_p \mid p \in \mathcal{I}\}$, such that every valid assignment to the propositional variables $\mathcal{P}_x$ corresponds to an element in the correlation set $\mathcal{C}_x$. Additionally, just like $\mathcal{I}$, $\mathcal{P}_x$ can be further split as $\mathcal{P}_x = \bigcup_i \mathcal{K}^i_x \cup \mathcal{M}_x \cup \bigcup_t \mathcal{R}^t_x$. Example 1 gives an intuition of the introduced variable sets and correlation set encoding.

Example 1: Let $\mathcal{I} = \{s_0, s_1, m\}$ be the labeled input ports given by the user, where $s = s_0 \oplus s_1$ is a secret with shares $\mathcal{K}^0 = \{s_0, s_1\}$, and fixed uniformly random masks $\mathcal{M} = \{m\}$. Let $\mathcal{C}_x = \{\perp, s_1, s_0 \oplus m, s_0 \oplus s_1 \oplus m\}$ be a correlation set. Then $\mathcal{P}_x = \{x_{s_0}, x_{s_1}, x_m\}$ are the propositional variables used for encoding $\mathcal{C}_x$, where $\mathcal{K}^0_x = \{x_{s_0}, x_{s_1}\}$, and $\mathcal{M}_x = \{x_m\}$, and there are no random ports. The propositional variables in $\mathcal{P}_x$ are constrained in such a way that the only satisfying assignments for the propositional tuple (x_{s_0}, x_{s_1}, x_m) are $(\perp, \perp, \perp)$, $(\perp, \top, \perp)$, $(\top, \perp, \top)$, and $(\top, \top, \top)$. These assignments represent the elements of $\mathcal{C}_x$, where x_p indicates whether the port p appears in the current term of $\mathcal{C}_x$.

COCOALMA maps the correlation terms in $\mathcal{C}_x$ to satisfying assignments to the propositional variables $\mathcal{P}_x$ by translating the propagation rules from Table I into satisfiability constraints. However, in order to simplify the exposition, we only demonstrate how we encode the correlation set operations $\langle\cdot\rangle$, $\cup$, and $\otimes$, as well as the creation of a correlation set with only one element. All of the propagation rules from Table I can be obtained by applying different combinations of these individual encodings, e.g., the transient rule for linear gates is obtained by combining the encodings of $\langle\cdot\rangle$ and $\otimes$.

First off, the correlation set of an input port only contains the port itself. Therefore, we restrict all of its propositional variables that correspond to other ports to be $\perp$, whereas the propositional variable representing the port itself must be set to $\top$. More precisely, for a port p in clock cycle t, the propositional variables $\mathcal{P}_x$ are constrained with

$$x_{p^t} \wedge \bigwedge_{x_a \in \mathcal{P}_x, a \neq p^t} \neg x_a \,, \tag{2}$$

where only *random* input ports are different in each clock cycle and $p = p^t$ in all other cases.

Extending a correlation set $\mathcal{C}_x$ with the $\perp$ element, written as $\langle\mathcal{C}_x\rangle$, is required for the propagation rules of linear and non-linear operations. When translating this into constraints for propositional variables $\mathcal{P}_x$, COCOALMA introduces a new set of variables $\mathcal{P}'_x$ and a fresh propositional variable q. The SAT solver can pick the value of q freely. Depending on the choice, all propositional variables $\mathcal{P}'_x$ are forced to equal their corresponding variables in $\mathcal{P}_x$ or forced to be $\perp$. We write this constraint as

$$\bigwedge_{x_a \in \mathcal{P}_x,\, x'_a \in \mathcal{P}'_x} x'_a \leftrightarrow (q \wedge x_a)\,. \tag{3}$$

All satisfying assignments of $\mathcal{P}'_x$ correspond to elements of the correlation set $\langle\mathcal{C}_x\rangle$. Each time the propagation rules in Table I use the $\langle\cdot\rangle$ operator, we introduce the variables $\mathcal{P}'_x$ and q and apply the given constraint.

Encoding the propagation rule for multiplexers requires a similar constraint when representing the union of two correlation sets. Given the correlation set $\mathcal{C}_z = \mathcal{C}_x \cup \mathcal{C}_y$, we introduce corresponding propositional variables $\mathcal{P}_z$ and a fresh propositional variable q. We subsequently constrain the introduced propositional variables with

$$\bigwedge_{z_a \in \mathcal{P}_z,\, x_a \in \mathcal{P}_x,\, y_a \in \mathcal{P}_y} z_a \leftrightarrow ((q \wedge x_a) \vee (\neg q \wedge y_a))\,, \tag{4}$$

where whenever $q = \top$ an element of $\mathcal{C}_x$ is encoded, and otherwise an element of $\mathcal{C}_y$. This encoding ensures that $\mathcal{C}_z$ contains all elements of $\mathcal{C}_x$ and $\mathcal{C}_y$, even if they are duplicates.

Finally, COCOALMA encodes the element-wise exclusive-or of two correlation sets $\mathcal{C}_z = \mathcal{C}_x \otimes \mathcal{C}_y$ using their corresponding propositional variables and a straightforward equivalence encoding

$$\bigwedge_{z_a \in \mathcal{P}_z,\, x_a \in \mathcal{P}_x,\, y_a \in \mathcal{P}_y} z_a \leftrightarrow (x_a \oplus y_a)\,. \tag{5}$$

Unlike the encoding of unions, no additional fresh propositional variables are necessary as there is no choice involved.

The constraints (2)-(5) only show how each of the propagation rules shown in Table I can be translated into SAT. COCOALMA needs an additional encoding for the conditions under which information leakage occurs. With correlation sets, we check whether there is an element of the correlation set where all shares of a secret are present, without being hidden by uniformly random values, such as fixed masks, random input ports, or shares of other secrets. Looking back at Example 1, we see that each time both shares s_0 and s_1 appear in a correlation term, they are masked by mask m. This means that the correlation set does not leak information about $s = s_0 \oplus s_1$. When checking this leakage property using the SAT encoding, we require two constraints.

First, we enforce that for each secret, either all shares are active, or all shares are inactive. Furthermore, we say that at least one secret must be active in order to have a leak. We encode this property by introducing one fresh propositional variable k_i for each secret and constraining them with

$$\left(\bigvee_i k_i\right) \wedge \bigwedge_i \bigwedge_{x_s \in \mathcal{K}^i_x} k_i \leftrightarrow x_s\,. \tag{6}$$

The first conjunct guarantees that at least one of the secrets is present in the correlation term. The rest of the expression ensures that either all shares of a secret are active in a correlation term, or none of them are, which is necessary since shares of incomplete secrets are uniformly random.

Second, we enforce that no masks appear in the correlation term, so the secrets are not *hidden* by uniformly random values, as discussed in Example 1. We represent this in the SAT encoding as

$$\left(\bigwedge_{x_m \in \mathcal{M}_x} \neg x_m\right) \wedge \left(\bigwedge_t \bigwedge_{x_r \in \mathcal{R}^t_x} \neg x_r\right)\,, \tag{7}$$

which ensures that a satisfying solution must assign all the variables representing masks and random values with $\perp$.

Constraints (6) and (7) go hand in hand, and both are required when testing whether a given correlation set leaks information about the secrets. When checking the security of a circuit in one of the supported security models, COCOALMA determines the observations an attacker can make, where each observation is made up of multiple correlation sets. For the *software probing model*, COCOALMA takes all the d-tuples $\mathcal{O}$ of probing locations (g, t) and tests the non-linear combination of their stable correlation sets

$$\bigotimes_{(g,t) \in \mathcal{O}} \left\langle S^t_g \right\rangle\,, \tag{8}$$

where g is the chosen gate, and t is the chosen clock cycle. The same applies to the *time-constrained probing model*, where COCOALMA checks the transient correlation sets T^t_g instead. In contrast, for the full *hardware probing model*, the probing locations $\mathcal{O}$ are a d-tuple of gates g instead, and concern all the clock cycles t for the given gates. Therefore, COCOALMA must check the correlation set

$$\bigotimes_{g \in \mathcal{O}} \bigotimes_t \left\langle T^t_g \right\rangle\,, \tag{9}$$

which significantly increases the observations an attacker can make. For example, using a register to store one share of a secret early in the computation and store the other share later in the computation would still allow an attacker to reconstruct the secret. Naturally, longer executions of a circuit get progressively harder to verify.

C. Encoding Optimizations

Although the shown SAT encoding is sufficient for showing whether the circuit leaks information about the processed secrets, the size of the produced constraints and formulas is unnecessarily large. In this section, we present some of the optimizations that dramatically reduce the effort of showing that a masked hardware circuit is secure.

Variable elimination. The sets of propositional variables $\mathcal{P}_x$ often include variables constrained through unit clauses, so their assignment is predetermined and equal in all satisfying solutions. Constraint (2) is an example of such a situation. Building constraints for such variables is unnecessary, and they can be removed entirely, substantially reducing the size of formula given to the SAT solver. In practice, COCOALMA implements this by storing $\mathcal{P}_x$ as a dictionary of propositional variables, as well as a set of variables trivially set to $\top$. All variables from $\mathcal{P}_x$ that are not present are known to have the value $\bot$. Consequently, whenever creating any of the shown constraints (3)–(7), we first check for trivial simplifications using the properties of logic operators. Although this optimization might seem superficial, it single-handedly reduces the number of variables and clauses by anywhere between 90% and 98% for the probing verification problems we have investigated so far. Notably, this optimization does not reduce the complexity of the queries given to the SAT solver, as solvers usually detect unit clauses anyway, but instead significantly reduces the memory consumption. Without this optimizations, verifying the probing security of longer executions would not be possible because the formula would not fit into memory.

Covering sets. Due to the nature of the propagation rules from Table I, some correlation sets are supersets of others. Take the propagation rules for non-linear gates as an example. For gate $f = a \wedge b$, the stable correlation set is computed as $S_f^t = \langle S_a^t \rangle \otimes \langle S_b^t \rangle = \{\bot\} \cup S_a^t \cup S_b^t \cup (S_a^t \otimes S_b^t)$, which implies that $S_a^t \subseteq S_f^t$ and $S_b^t \subseteq S_f^t$. Consequently, it is sufficient to perform the security checks for S_f^t, ignoring both S_a^t and S_b^t because their elements are already *covered*. For element-wise exclusive-or operations like $\mathcal{C}_z = \mathcal{C}_x \otimes \mathcal{C}_y$, the resulting set $\mathcal{C}_z$ covers $\mathcal{C}_x$ whenever $\bot \in \mathcal{C}_y$, and $\mathcal{C}_y$ whenever $\bot \in \mathcal{C}_x$. It turns out that in the *software probing model*, we only need to check gates that are inputs to XOR gates, selectors of a multiplexer, inputs to a register, and circuit outputs. In the *time-constrained probing model*, we only check register inputs and circuit outputs because in that model linear gates behave non-linearly due to glitches. In the full *hardware probing model*, the covering properties are slightly more complex, and we check all gates that have at least one clock cycle where another gate does not cover them.

Table II
SIMPLIFICATION RULES FOR STABLE CORRELATION SETS

Gate type	f	Stable set $\mathcal{C}_f$	f	Stable set $\mathcal{C}_f$
Linear	$a \oplus \bot$	$\mathcal{C}_a$	$a \oplus \top$	$\mathcal{C}_a$
Non-linear	$a \wedge \bot$	–	$a \wedge \top$	$\mathcal{C}_a$
	$a \vee \bot$	$\mathcal{C}_a$	$a \vee \top$	–
Multiplexer	$\bot \,?\, a : b$	$\mathcal{C}_b$	$\top \,?\, a : b$	$\mathcal{C}_a$
	$c \,?\, \bot : b$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_b \rangle$	$c \,?\, \top : b$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_b \rangle$
	$c \,?\, a : \bot$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_a \rangle$	$c \,?\, a : \top$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_a \rangle$

IV. SIMULATIONS

Although the method presented in Section III is sufficient to check the security of a masked implementation in the supported probing models, it does not consider how the control signals change over time. As mentioned in the introduction, COCOALMA uses simulations to obtain information about the exact values of control signals and subsequently uses them to simplify the correlation sets accordingly.

In the hardware probing model, all values marked as *sensitive*, *i.e.*, secret shares, mask registers and random input ports, are assumed to be uniformly random. This is a requirement for the execution environment, in this case the testbench, which performs the secret sharing steps and includes a random number generator that drives the random input ports in each clock cycle. In any reasonable probing model, the attacker can only control the values of un-shared plaintext values, and we assume they can request an unlimited number of encryptions for the DPA attack. If the attacker were able to mess with the random number generator of the environment, they would be able to break any conceivable masking scheme, so this is out-of-scope in the hardware probing model.

Other input signals, such as control signals, which marked as *public* are assumed to be independent of the secrets and masks processed in the hardware circuit, so their values can be taken directly from a circuit simulation. Since their values are known, COCOALMA uses them to perform simplifications while applying the propagation rules. Consider the gate $f = a \wedge b$, where a is a public value and b has a correlation set $\mathcal{C}_b$. Because COCOALMA knows the value of a, f is simplified accordingly. If $a = \bot$, then we know that $f = \bot$ independently of b, meaning that f is also a public value and does not need a correlation set. Similarly, if $a = \top$, we know that $f = b$, and we can reuse the correlation set as $\mathcal{C}_f = \mathcal{C}_b$. Table II defines analogous simplifications for all propagation rules with multiple inputs when the constant signal is stable. Using the simulated execution of the circuit and the labeling provided by the user, each gate g at each clock cycle t is classified as either being a control signal or having a correlation set, but never both. Empty entries in Table II indicate that the gate does not have a correlation set and is instead declared a control signal.

A. Signal Stability

Unlike with stable correlation sets, applying simplifications based on the simulation trace is not straightforward for transient correlation sets, where COCOALMA must also consider

Table III
SIGNAL STABILITY COMPUTATIONS

Gate type of f		Computation of $st(f)$ in current clock cycle
Constant	$\bot$ or $\top$	$\top$
Input Port	p	$\neg\, cr(p)$
Negation	$\neg a$	$st(a)$
Register	$\Leftarrow_R a$	$\neg\, cr'(a) \wedge (vl'(a) \leftrightarrow vl'(f))$
Linear	$a \oplus b$	$st(a) \wedge st(b)$
Non-linear	$a \wedge b$	$st(a) \wedge \neg\, vl(a) \vee st(b) \wedge \neg\, vl(b) \vee st(a) \wedge st(b)$
	$a \vee b$	$st(a) \wedge vl(a) \vee st(b) \wedge vl(b) \vee st(a) \wedge st(b)$
Multiplexer	$c\,?\,a\,:\,b$	$st(c) \wedge (vl(c) \wedge st(a) \vee \neg\, vl(c) \wedge st(b)) \vee$ $\vee\, st(a) \wedge st(b) \wedge (vl(a) \leftrightarrow vl(b))$

Table IV
VERIFICATION RESULTS FOR TWO VERSIONS OF PRINCE-TI

Algorithm	#Sec.	#Rand.	#Rnds.	#Cyc.	SW	TC	HW
PRINCE-TI	192	48	1	3	✓ 0.72 s	✗ 1.97 s	✗ 2.43 s
PRINCE-TI	192	192	1	3	✓ 3.37 s	✓ 7.21 s	✓ 11.57 s
PRINCE-TI	192	192	2	5	✓ 187.8 s	✓ 150.6 s	✓ 236.9 s
PRINCE-TI	192	192	3	7	✓ 0.77 h	✓ 3.80 h	✓ 17.92 h
AES-DOM	256	46	1	21	✓ 195.3 s	✓ 1.82 h	✓ 2.89 h

glitches. Glitches are hardware phenomena that behave like temporary faults while switching values. A gate $f = a \odot b$ will pass on a's value if its signal arrives at f before the new signal of b. After both signals arrived, the fault is corrected, and f becomes the value it is supposed to have. Ultimately, the signal must be stable at the end of a clock cycle, when the clock triggers the registers and synchronizes the computation.

However, there are certain conditions when a gate cannot experience a glitch, e.g., when the values a and b come directly out of a register and do not change from the previous clock cycle. In that particular case, even though the signal timings are different, the value transmitted through the wires did not change the entire time, and no glitching is possible. As a result, even the signal produced by f would be stable and glitch-free. This property recursively propagates throughout the whole circuit and allows us to determine which values can be used for the simplifications shown in Table II, even for transient correlation sets.

COCOALMA uses the concrete values of a simulation trace to determine the glitching behavior of public values such as control signals. Assume the same situation as before, with $f = a \wedge b$, where a is a public value and b might correlate with masks or shares, and thus, has a correlation set $\mathcal{C}_b$. Knowing whether f can forward b is crucial, as it might lead to an information leak in a later part of the circuit. If $a = \bot$ and its signal is stable, meaning it cannot produce glitches, then f is a public value with $f = \bot$. Therefore, a being a stable public signal set to $\bot$ effectively stops the propagation of a correlation set from b to f. In the rest of this section, we outline a recursive method for determining whether a signal is stable in a given clock cycle.

In the following exposition, we introduce three predicates that help define the algorithm computing the signal stability. We use the $st(x)$ predicate to say that the signal x is stable. The predicate $cr(x)$ is true whenever the signal x is associated with a transient correlation set. Finally, predicate $vl(x)$ represents the value of signal x taken from the execution trace. All three predicates also have a version that applies to the previous clock cycle: $st'(x)$, $cr'(x)$, and $vl'(x)$. The rules computing the stability of any given signal f are shown in Table III. All values of the predicates are computed directly, and none of them are given to the SAT solver.

First, all input ports are held stable by the environment. That is, another circuit that controls the input ports must keep their signals stable and avoid glitches. Since public signals and signals with correlation sets are mutually exclusive in COCOALMA, an input port is only considered stable when it does not have a correlation set. Similarly, the output of a register is stable if the register does not change its value from the previous cycle and does not have a correlation set associated with its input. If the value did change, we consider the signal unstable because it can cause glitches in gates connected to it during the clock-cycle transition. Linear gates such as XOR are only stable if both of their inputs are stable. If one of the inputs produces a glitch, then an XOR would forward it to all gates it is connected to since the other signal cannot stop it.

Non-linear gates such as AND (OR) can remain stable even if one of their inputs produces glitches. If at least one of the inputs of an AND (OR) gate is stable at $\bot$ ($\top$), then no change or glitch in the other input can make it unstable. Otherwise, the output of an AND (OR) gate is only stable if both of its inputs are also stable. The conditions under which a multiplexer is stable are similar. For instance, if selector c is stable with the value $\top$ ($\bot$), then the output of the multiplexer is stable if and only if the selected input a (b) is stable. In contrast, if selector c is not stable, the output is only stable if the inputs a and b are stable and have equivalent values.

V. CASE STUDIES

In this section, we investigate the probing security of the masked hardware implementations PRINCE-TI [6] and AES-DOM [16]. In particular, we analyze the complexity of verifying round-reduced versions in all three of the supported probing models. Additionally, we demonstrate how COCOALMA's debugging functionalities allow us to identify potential issues and fix them accordingly. All experimental results shown in Table IV were captured on a notebook with the Intel Core i7-8550U 1.8GHz CPU and 16 GiB of RAM.

A. *Verifying PRINCE-TI*

PRINCE is a state-of-the-art lightweight block cipher. It is designed with hardware implementations in mind, so that ideally, the entire encryption process can be done in one clock cycle [5] when no masking is applied. PRINCE takes as input a 64-bit plaintext block and encrypts it with a 128-bit key. The encryption process consists of two phases with six rounds each. In the first phase, the first round adds the round key onto the data block, whereas the other five rounds apply a 4-bit S-Box, an affine transformation, and then mix the round key into the data block. After the first phase, the data block is transformed using the 4-bit S-Box, another affine transformation, and the inverse 4-bit S-Box, before starting the second phase. In the second phase, each round applies the inverse operations performed in the rounds of the first phase, meaning that the first five rounds add the round key, apply the inverse affine transformation followed by the inverse 4-bit S-Box. The last round of the second phase only adds the round key to the data block.

Unlike the unmasked version of Prince, the threshold implementation Prince-TI [6] cannot be completed in one clock cycle. This restriction is due to the re-sharing phase present in threshold implementations, which requires additional synchronization to prevent leakage caused by glitches. For first-order probing security, the implementation splits all the plaintext and key bits into two shares and treats them as secrets. Prince-TI uses random inputs to re-share the outputs of its sixteen 4-Bit S-Boxes, where each S-Box requires twelve random bits. In the official implementation, this process is optimized in such a way that four S-Boxes share the same randomness, so the re-sharing only requires a total of 48 random bits.

The first row of Table IV shows the results produced by CocoAlma, where 192 (*i.e.*, 128 key bits and 64 plaintext bits) pairs of ports are labeled as shares of secrets, and 48 ports are labeled as coming from a random number generator. The first round of the cipher needs three clock cycles to complete since we first need to load the inputs into internal registers and start the encryption. Within one second, CocoAlma has proven that the implementation is secure in the *software probing model* (SW), indicated with (✔) in Table IV. However, CocoAlma claims it found a leak (✖) in the *time-constrained probing model* (TC) in the third clock cycle and provides us with debugging information.

B. Debugging Information

After finding a leak in a hardware circuit, CocoAlma attempts to simplify the leaking correlation. For example, CocoAlma could report that the output of a gate correlates with the linear combination of many secrets. This information, while correct, is often not useful for a designer because looking through the implementation and tracking the data dependencies of so many secret bits is extremely cumbersome. Therefore, CocoAlma attempts to minimize the number of secrets in the leaking correlation term. In particular, we go through all secret bits and greedily assume that the leaking correlation term does not contain them but still leaks information. If the SAT solver returns UNSAT, we know that the investigated secret must appear in the correlation term. At the end of this procedure, CocoAlma has produced a minimized example of a leaking correlation term.

Next, CocoAlma provides a *leakage graph*, which allows the designer to visualize the structure of the leaking part of the circuit. In particular, the leakage graph highlights the leaking gates and only includes gates that influence the leak. We perform this graph minimization by starting at the leaking gates and computing their *cone of influence*.

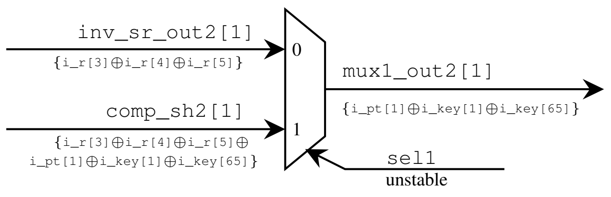
inv_sr_out2[1]
{i_r[3]⊕i_r[4]⊕i_r[5]}
0
comp_sh2[1]
{i_r[3]⊕i_r[4]⊕i_r[5]⊕ i_pt[1]⊕i_key[1]⊕i_key[65]}
1
mux1_out2[1]
{i_pt[1]⊕i_key[1]⊕i_key[65]}
sel1
unstable

Figure 2. The Prince-TI leakage found with CocoAlma. Signal names are shown on top of lines, whereas the problematic correlation term or signal stability is shown below.

Finally, CocoAlma produces a *leakage trace* where the correlation terms of all relevant correlation sets are displayed. In particular, we take the model produced by the SAT solver and show the ports $p \in \mathcal{I}$ whose corresponding propositional variables in $\mathcal{P}_x$ are assigned to $\top$, indicating they are part of the correlation term. The designer can combine this information with the leakage graph to deduce the cause of the leak.

C. Debugging PRINCE-TI

In the particular case of Prince-TI, we have identified the leak at multiplexer `mux1_out2[1]`, as shown in Figure 2. Here, the control signal `sel1` determines whether the output is the inverse of the shift rows operation `inv_sr_out2[1]`, or the compression operation `comp_sh2[1]`. Here, a glitch on the control signal `sel1` causes the multiplexer to forward both inputs in the third clock cycle. Unfortunately, `inv_sr_out2[1]` correlates to the uniformly random value $r =$ `i_r[3]`$\oplus$`i_r[4]`$\oplus$`i_r[5]`, whereas `comp_sh2[1]` correlates with $r \oplus$ `i_pt[1]`$\oplus$`i_key[1]`$\oplus$`i_key[65]`. Observing these two values allows an attacker to compute `i_pt[1]`$\oplus$`i_key[1]`$\oplus$`i_key[65]`, breaking the security guarantees promised by masking schemes.

Although the leakage is observable at `mux1_out2[1]`, its root cause is somewhere else. Under closer inspection of the *leakage trace* and *leakage graph*, we see that the shift rows operation, in combination with glitches, causes a forwarding of the random bits used to re-share the thirteenth S-Box, making them observable at `inv_sr_out2[1]`. Since the same random bits are used to re-share the first S-Box, which eventually leads to `comp_sh2[1]`, the random bits cancel out at the multiplexer. Ultimately, the reuse of random bits causes a leak in the presence of glitches. We fix this by increasing the size of the random input `i_r` from 48 to 192 bits, and avoiding the reuse of random inputs for the re-sharing of S-Box outputs. The second and third row of Table IV show the verification results for the fixed version of Prince-TI, where we were able to verify up to two rounds of the cipher in under four minutes.

D. Verifying AES-DOM

Rijndael, better known as the *Advanced Encryption Standard* (AES), is an extremely popular, secure, and widely adopted block cipher [8]. The 128-bit version of AES takes as input a 128-bit plaintext and encrypts it through ten rounds using a 128-bit key. First, the cipher adds the initial secret key to the plaintext to create the cipher's state and then expands the key into ten individual round keys. The first nine rounds apply the S-Box to each state byte, re-order the bytes, apply a linear transformation to 32-bit chunks, and mix the state with the round key. The last round does not apply the linear transformation as it does not contribute to security.

AES is not intended for masked implementations because it has a highly non-linear S-Box that is applied sixteen times per round. In order to minimize the used design area, masked AES implementations opt for only one S-Box module that is sequentially fed new bytes each clock cycle [25], [16].

We have analyzed the probing security of the DOM-protected [16] implementation of AES by Gross et al. in all three security models. The open-source implementation of AES-DOM[4] is written in VHDL and not in Verilog, so it is not directly compatible with our verification flow. However, due to the modularity of COCOALMA, we can produce a netlist with another synthesis flow, e.g., GHDL[5], and extend it with a compatibility wrapper in Verilog so we can use Verilator for the *tracing* step of the original verification flow depicted in Figure 1. Although this is convenient, it is not strictly required, and COCOALMA also supports execution traces produced by other simulators in VCD format.

Executing the first round of the cipher requires one cycle of setup and twenty computation cycles. Notably, because of the parallelism in hardware designs, AES-DOM computes the linear operations of the first round just-in-time for their use as S-Box inputs in the second round. Therefore, the first 21 cycles only include the key addition, sixteen S-Box applications, and the byte re-ordering. The implementation processes 256 secrets, that is, 128 key bits and 128 plaintext bits. In each clock cycle, the AES-DOM consumes 46 uniformly random bits, yielding a total of 966 random bits for the first round of the cipher. The last column of Table IV shows the verification results for the first round of AES-DOM. The verification was successful in all three probing models, and since the AES-DOM implementation is more complex than PRINCE-TI, it naturally takes longer to verify. COCOALMA only takes about three hours to verify that the implementation of AES-DOM is secure in the hardware probing model.

VI. Related Work

The formal verification of power analysis countermeasures is a well-established research field [1], [2], [4], [13], [10], [11], [19]. The community has been investigating two fundamentally different principles. On the one hand, there are approximative methods like those used in REBECCA [4], `maskVerif` [2], and COCOALMA. In contrast to REBECCA and COCOALMA, `maskVerif` opts for a language-based verification approach, tracks the symbolic representation of probing locations, and simulates the observations an attacker can make using uniformly random values. On the other hand, model counting methods inspect the truth table of a given function and check whether the correlation strength is zero for all secret values. Tools such as `QMVerif` [10] and `QMSInfer` [11] apply these methods to overcome the shortcomings of heuristics used in faster approximative methods. Similarly, probability-distribution tracking approaches such as SILVER [19] (implicitly) rely on model counting to determine the distribution type for any possible observation an attacker can make.

To our knowledge, `maskVerif` and SILVER were not used for stateful hardware verification. The authors of `QMVerif` and `QMSInfer` claim they support stateful hardware verification, but the tools are not open-source, so we could not replicate their results.

VII. Future Work

The current version of COCOALMA is a significant improvement over its predecessor REBECCA [4]. However, there are still open questions that could yield performance improvements or usability improvements.

The model of glitches used in COCOALMA seems too conservative, but we have no empirical evidence to the contrary. In particular, we assume that glitches are unpredictable and can forward any combination of the new and old signal values, even constants. This assumption might be too strict, and some combinations would not be observable in a power trace. Similarly, we assume the worst-case interaction between transition and glitch leakage, which might also be unnecessarily cautious. Eliminating these overly paranoid precautions would single-handedly reduce the verification complexity. Another avenue for increasing the scalability would be to consider implementation modules separately and tie the individual proofs together using composability notions [2].

VIII. Conclusion

Although COCOALMA was originally designed for verifying software in the *time-constrained probing model*, it can also verify stateful hardware circuits in the *hardware probing model*. COCOALMA improves upon REBECCA in terms of scope and verification capabilities. It supports more security models, includes an elegant correlation-set encoding, supports circuit simulation, and uses it throughout the verification. The native support for stateful verification allows a tighter integration into the design flow, and as demonstrated with PRINCE-TI and AES-DOM, COCOALMA can be applied to industry-scale designs. We have successfully identified a leakage location in PRINCE-TI, which cannot be found by only analyzing the PRINCE-TI S-Box, as it requires the full context of the cipher's implementation. Through the debugging support provided by COCOALMA, we found the cause of the information leakage and fixed it by adding more random inputs. Furthermore, we have also demonstrated the modularity and adaptability of COCOALMA by verifying an AES-DOM design that uses an entirely different synthesis flow in another HDL language.

Overall, we think COCOALMA is an excellent addition to any synthesis flow and can be used for the early detection of mistakes.

[4]https://github.com/hgrosz/aes-dom

[5]https://github.com/ghdl/ghdl-yosys-plugin

References

[1] Arribas, V., Nikova, S., Rijmen, V.: VerMI: Verification tool for masked implementations. In: ICECS 2018. pp. 381–384 (2018)

[2] Barthe, G., Belaïd, S., Cassiers, G., Fouque, P., Grégoire, B., Standaert, F.: maskverif: Automated verification of higher-order masking in presence of physical defaults. In: ESORICS 2019. pp. 300–318 (2019)

[3] Biere, A.: CaDiCaL at the SAT Race 2019. In: SAT Race 2019. pp. 8–9. University of Helsinki (2019)

[4] Bloem, R., Groß, H., Iusupov, R., Könighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. In: EUROCRYPT 2018 (2018)

[5] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: ASIACRYPT (2012)

[6] Bozilov, D., Knezevic, M., Nikov, V.: Optimized threshold implementations: Securing cryptographic accelerators for low-energy and low-latency applications. IACR (2018)

[7] Chellam, M.B., Natarajan, R.: AES hardware accelerator on FPGA with improved throughput and resource efficiency. AJSE (2018)

[8] Daemen, J., Rijmen, V.: Aes proposal: Rijndael (1999)

[9] Dhooghe, S., Nikova, S., Rijmen, V.: Threshold implementations in the robust probing model. In: TIS@CCS 2019. pp. 30–37 (2019)

[10] Gao, P., Xie, H., Zhang, J., Song, F., Chen, T.: Quantitative verification of masked arithmetic programs against side-channel attacks. In: TACAS (2019)

[11] Gao, P., Zhang, J., Song, F., Wang, C.: Verifying and quantifying side-channel resistance of masked software implementations. ACM Trans. Softw. Eng. Methodol. pp. 16:1–16:32 (2019)

[12] Ghosh, S., Zhao, L., Misoczki, R., Sastry, M.R.: Ultra-lightweight cryptography accelerator system. Tech. rep., Intel Corporation (2018), patent number: US20180183573A1

[13] Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-design and co-verification of masked software implementations on cpus. Tech. rep., IACR Cryptology ePrint Archive report 2020/1294 (2020)

[14] Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Handbook of satisfiability (2009)

[15] Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. CCS (2016)

[16] Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In: TIS@ CCS. p. 3 (2016)

[17] Groß, H., Mangard, S., Korak, T.: An efficient side-channel protected aes implementation with arbitrary protection order. In: RSA (2017)

[18] Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: CRYPTO 2003. pp. 463–481 (2003)

[19] Knichel, D., Sasdrich, P., Moradi, A.: SILVER - statistical independence and leakage verification. In: ASIACRYPT (2020)

[20] Kruse, J., Schinianakis, D.: A high-throughput, low area implementation of PRINCE algorithm for industrial IoT. Tech. rep., Bell Labs, Nokia (2017), https://www.bell-labs.com/institute/publications/itd-17-57329p/

[21] Mangard, S.: A simple power-analysis (SPA) attack on implementations of the AES key expansion. In: ICISC (2002)

[22] Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)

[23] Messerges, T.S., Dabbish, E.A.: Investigations of power analysis attacks on smartcards. In: USENIX Smartcard (1999)

[24] Moos, T., Moradi, A., Schneider, T., Standaert, F.: Glitch-resistant masking revisited or why proofs in the robust probing model are needed. TCHES pp. 256–292 (2019)

[25] Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of aes. In: EUROCRYPT 2011 (2011)

[26] Örs, S.B., Gürkaynak, F.K., Oswald, E., Preneel, B.: Power-analysis attack on an ASIC AES implementation. In: ITCC (2004)

[27] Quisquater, J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In: E-smart 2001. pp. 200–210 (2001)

[28] Snyder, W.: Verilator, https://www.veripool.org/wiki/verilator, https://www.veripool.org/wiki/verilator. Retrieved on July 10th, 2020

[29] Valiant, L.: The complexity of computing the permanent. Theoretical Computer Science (1979)

[30] Wang, Y., Ha, Y.: FPGA-based 40.9-gbits/s masked aes with area optimization for storage area network. IEEE TCAS II (2013)

[31] Wolf, C., Glaser, J.: Yosys — a free verilog synthesis suite. Proceedings of Austrochip (2013)