Prover - Toward More Efficient Formal Verification of Masking in Probing Model

Feng Zhou[1,2,3], Hua Chen[†2], Limin Fan[2]

[1] University of Chinese Academy of Sciences, Beijing, China
[2] TCA Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing, China
zhoufeng2021@iscas.ac.cn, chenhua@iscas.ac.cn, fanlimin@iscas.ac.cn
[3] Zhongguancun Laboratory, Beijing, China

Abstract. In recent years, formal verification has emerged as a crucial method for assessing security against Side-Channel attacks of masked implementations, owing to its remarkable versatility and high degree of automation. However, formal verification still faces technical bottlenecks in balancing accuracy and efficiency, thereby limiting its scalability. Former tools like maskVerif and CocoAlma are very efficient but they face accuracy issues when verifying schemes that utilize properties of Boolean functions. Later, SILVER addressed the accuracy issue, albeit at the cost of significantly reduced speed and scalability compared to maskVerif. Consequently, there is a pressing need to develop formal verification tools that are both efficient and accurate for designing secure schemes and evaluating implementations. This paper's primary contribution lies in proposing several approaches to develop a more efficient and scalable formal verification tool called Prover, which is built upon SILVER. Firstly, inspired by the auxiliary data structures proposed by Eldib et al. and optimistic sampling rule of maskVerif, we introduce two reduction rules aimed at diminishing the size of observable sets and secret sets in statistical independence checks. These rules substantially decrease, or even eliminate, the need for repeated computation of probability distributions using Reduced Ordered Binary Decision Diagrams (ROBDDs), a time-intensive procedure in verification. Subsequently, we integrate one of these reduction rules into the uniformity check to mitigate its complexity. Secondly, we identify that variable ordering significantly impacts efficiency and optimize it for constructing ROBDDs, resulting in much smaller representations of investigated functions. Lastly, we present the algorithm of Prover, which efficiently verifies the security and uniformity of masked implementations in probing model with or without the presence of glitches. Experimental results demonstrate that our proposed tool Prover offers a better balance between efficiency and accuracy compared to other state-of-the-art tools (IronMask, CocoAlma, maskVerif, and SILVER). In our experiments, we also found an S-box that can only be verified by Prover, as IronMask cannot verify S-boxes, and both CocoAlma and maskVerif suffer from false positive issues. Additionally, SILVER runs out of time during verification.

Keywords: Side-Channel Attacks · Masking · Formal Verification · Glitch-Extended Probing Security · Reduced Ordered Binary Decision Diagrams

1 Introduction

Cryptographic algorithms play a crucial role in our daily lives, being implemented in cryptographic devices widely deployed across various applications such as smart cards and IoT (Internet of Things) systems. In recent years, Side-Channel Attacks (SCA),

[†]Hua Chen is the corresponding author.

including timing attacks [Koc96] and power analysis [KJJ99], have emerged as potent threats against cryptographic modules. To mitigate the risks posed by SCAs, numerous countermeasures have been proposed, among which masking [ISW03, Tri03] stands out as one of the most effective techniques. Based on the concept of secret sharing, the security of masking schemes can be proven theoretically under reasonable assumptions. Ishai et al. [ISW03] demonstrated the security of their schemes within their proposed *d-probing model* (or standard probing model), which has since been widely adopted by subsequent works. However, it has been shown that certain physical anomalies, such as glitches, transitions, and couplings, can compromise the leakage assumptions in the d-probing model [FGP+18]. In the realm of hardware masking, a significant challenge arises as these schemes are susceptible to vulnerabilities in the presence of glitches [MPG05, MPO05]. To address this challenge, threshold implementations (TI) [NRR06] have emerged, providing inherent resistance to glitches through the fulfillment of three key properties: correctness, incompleteness, and uniformity. Following TI, numerous masking techniques against glitches have been proposed [RBN+15a, GMK16, SM21a].

To formally define (or verify) the security of masking in the presence of glitches, Bloem et al. [BGI+18] and Faust et al. [FGP+18] independently extended the d-probing model, incorporating glitches, and introduced a more robust model known as the *glitch-extended probing model*.

However, the theoretical security of masking schemes in formal models does not directly ensure the practical security of corresponding implementations. Hence, it becomes imperative to verify and assess the effectiveness of masking countermeasures. Considerable effort has been invested in verifying software implementations [MOPT12, BRNI13, EWS14, BBD+15, ZGSW18, BGR18, GXSC21], yielding more sophisticated approaches. In contrast, the range of existing frameworks encompassing verification under the glitch-extended probing model is quite limited. Based on the adopted approaches, research on such frameworks can be categorized into the following three types.

Bloem et al. [BGI+18] introduced the first formal verification tool, REBECCA, designed to assess the security of hardware implementations. Their approach leverages the spectral characteristics of Boolean functions. Specifically, if a nonzero spectral coefficient exists between any Boolean function defined over the circuit outputs and a linear combination of sensitive variables, the implementation is deemed insecure. However, due to the considerable time overhead required to compute spectral coefficients, REBECCA resorts to certain approximations via SAT encoding, resulting in false positives. Additionally, the inevitable use of SAT solvers leads to inefficiencies and limited scalability. Subsequent developments of REBECCA, CocoAlma [GHP+21, HB21], demonstrate improved usability and performance.

In the realm of language-based verification methods (the second approach), Barthe et al. [BBC+19] extended their techniques proposed in [BBD+15] to address hardware masking, creating a unified framework known as maskVerif. maskVerif accommodates various security notions such as standard probing security, (Strong) Non-Interference (NI/SNI) [BBD+16], as well as robust security notions under the d-probing model with glitches or transitions. Compared to the REBECCA tool, maskVerif exhibits high efficiency. However, owing to the conservative nature of language-based methods, false positives are also inevitably encountered.

The third approach is rooted in the concept of statistical independence. Building upon the efforts to consolidate security notions [DBR19], Knichel et al. [KSM20] reformulated the concepts of probing security, (Strong/Probe Isolating) Non-Interference (NI/SNI/PINI) from the perspective of probability distributions. Based on this approach, the SILVER tool was developed, surpassing the capabilities of maskVerif by incorporating verification of the PINI security notion [CS20]. During verification, SILVER constructs Reduced Ordered Binary Decision Diagrams (ROBDDs) for every possible combination of observations in the

circuit to compute probability distributions. However, as ROBDDs have finite capabilities in representing Boolean functions with numerous input variables, SILVER faces limitations in terms of efficiency and scalability. Particularly when larger masked circuits are involved, SILVER exhibits significant inefficiencies.

In addition to leakage caused by glitches, several studies also address formal verification for transitional leakage in the robust probing model, including CocoAlma [GHP+21, HB21], SILVER [MKSM22], and fullverif [CGLS20, MCS22]. In this paper, we focus on leakage caused by glitches and will not explore transitional leakage.

Another research area concerns the design and verification of masked implementations based on composable gadgets. These gadgets, such as robust SNI ISW multiplication [FGP+18], HPC1 and HPC2 [CGLS20], or HPC3 [KM22], often rely on security notions stronger than probing security, such as SNI [BBD+16] and PINI [CS20]. Tools like fullverif [CGLS20] are capable of verifying the complete designs of masked implementations based on the framework of PINI. There are also tools that verify security properties of standard gadgets under probing model or random probing model [Ajt11, DDF14], such as IronMask [BMRT22].

Indeed, formal verification of security in masked implementations has made notable progress. However, existing research results still grapple with challenges in balancing accuracy and efficiency. Specifically, current tools are either *fast but inaccurate* (such as maskVerif) or *accurate but slow* (like SILVER). For instance, consider verification under the glitch-extended probing model: maskVerif takes no more than one second to confirm the security of the first-order DOM implementation of the AES S-box [BBC+19], yet it inaccurately reports the secure implementation of $\mathcal{Q}_{12}^4$ in [BNN+15] as insecure [KSM20]. Moreover, several works, e.g., [SM21a], have asserted that maskVerif cannot verify the security of their constructions without fresh randomness. Meanwhile, SILVER correctly verifies the security of $\mathcal{Q}_{12}^4$ but takes 20 minutes to confirm the security of the DOM implementation on a significantly more powerful machine [KSM20]. This situation underscores the need for an approach that integrates efficient heuristic rules with accurate probability enumeration to achieve a better balance between accuracy and efficiency.

Our Contributions. In this paper, we tackle the low efficiency issue of SILVER and propose several methods to significantly enhance efficiency while maintaining accuracy, resulting in the formal verification tool, Prover. Firstly, inspired by the auxiliary data structures introduced in [EWS14], we are able to adopt the similar idea of optimistic sampling rule from maskVerif [BBD+15, BBC+19] in our work. This led us to introduce two reduction rules aimed at significantly decreasing the size of observation and secret sets. These rules are applicable in statistical independence checks under both standard and glitch-extended probing models, as well as in the verification of uniformity. By reducing the size of observation sets (and secret sets), which are exponentially related to complexity, the actual computational complexity is substantially reduced. In some cases, these rules even lead to the elimination of the observation set, rendering the expensive operation of constructing ROBDDs unnecessary. Secondly, we observe that the variable ordering of ROBDDs profoundly impacts SILVER's performance. Through analysis, we identify two potentially more efficient orderings and validate our findings through experiments. The optimized orderings prove to be much more efficient, particularly in larger masked circuits. Finally, we implement our approaches into Prover, a tool built upon SILVER. We also conducted extensive experiments to compare Prover to other state-of-the-art tools: IronMask, CocoAlma, maskVerif, and SILVER. Experimental results illustrate that Prover achieves a superior balance between efficiency and accuracy compared to the other tools. Moreover, with IronMask beyond the scope of S-box verification, Prover successfully verified an S-box implementation that SILVER failed to complete within the allotted time, while CocoAlma and maskVerif encountered false positive issues.

2 Preliminaries

2.1 Symbols and Notations

The symbols and notations used in this paper are shown in Table 1.

Table 1: The symbols and notations used in this paper

Symbols / Notations	Meaning
$\mathrm{GF}_2, \mathrm{GF}_2^n$	binary field, vectorial space over binary field
$x, \boldsymbol{x}$	a variable, a set of variables
α	a boolean value
$\boldsymbol{\alpha}$	a set of boolean values, also denoted by an integer $\sum_{i=0}^{\lvert\boldsymbol{\alpha}\rvert} \alpha_i \cdot 2^i$
$\uplus, \cap, \cup, \setminus$	disjoint set union, set intersection, set union, set difference
$\lvert S\rvert, \emptyset$	size of a set S, empty set
$\oplus, +$	exclusive-or, addition in binary field
$\wedge, \cdot$	and operation, multiplication in binary field
$\neg, \vee, \overline{\vee}, \overline{\wedge}, \overline{\oplus}$	negation, or, nor, nand, xnor operation in binary field
$\bigoplus_i x_i, \bigwedge_i x_i$	summation and production in binary field
$\boldsymbol{x} = \boldsymbol{\alpha}$	$\lvert\boldsymbol{x}\rvert = \lvert\boldsymbol{\alpha}\rvert$, and for $1 \le i \le \lvert\boldsymbol{x}\rvert$, $x_i = \alpha_i$
$\boldsymbol{x}^{\boldsymbol{\lambda}}(\boldsymbol{\lambda}\boldsymbol{x})$	a product(linear) combination $\bigwedge_i x_i^{\lambda_i}(\bigoplus_i x_i^{\lambda_i})$ of variables in $\boldsymbol{x}$
$\boldsymbol{x}', \boldsymbol{x}_{\bar{i}}$	subset of $\boldsymbol{x}$, $\boldsymbol{x} \setminus x_i$
$\Pr[A]$	probability of a event A
$Sh(x), Sh(\boldsymbol{x})$	the shares of variable x, the shares of variables in $\boldsymbol{x}$
$\mathcal{O}_d$	d-th order observation set, i.e., the union of observation set of d gates
(ni)supp(n)	the set of variables that appear in the expression of f_n
perf(n)	the set of perfect mask of observation function f_n

2.2 Probability Distributions of Boolean Variables

A Boolean random variable $x \in \mathrm{GF}_2$ can take on the values 0 or 1. A set of Boolean random variables $\boldsymbol{x}$ consists of Boolean variables.

First, we define the probability mass function of a Boolean random variable set.

Definition 1 (**Probability Mass Function**)**.** The probability mass function of a Boolean variable set $\boldsymbol{x}$ is defined as $p_{\boldsymbol{x}}(\boldsymbol{\alpha}) = \Pr[\boldsymbol{x} = \boldsymbol{\alpha}]$.

Given any two Boolean random variable sets, we can define their joint probability mass function.

Definition 2 (**Joint Probability Mass Function**)**.** The joint probability mass function of Boolean variable sets $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as $p_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \Pr[\boldsymbol{x} = \boldsymbol{\alpha}, \boldsymbol{y} = \boldsymbol{\beta}]$.

The relationship between the probability mass function and the joint probability mass function is $p_{\boldsymbol{x}}(\boldsymbol{\alpha}) = \sum_{\boldsymbol{\beta}} p_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

Based on the definition of the joint probability mass function of Boolean variable sets, we can define the statistical independence of two Boolean variable sets.

Definition 3 (**Statistical Independence**)**.** Two Boolean variable sets $\boldsymbol{x}$ and $\boldsymbol{y}$ are statistically independent if and only if for any possible combinations of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the equation $p_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = p_{\boldsymbol{x}}(\boldsymbol{\alpha}) p_{\boldsymbol{y}}(\boldsymbol{\beta})$ holds.

2.3 Masked Circuits

The hardware implementation of a masking scheme, a physical circuit, consists of combinational gates, registers, and wires. For convenience, both combinational gates and registers are referred to as gates in the rest of this paper. Each gate performs an operation on its input and outputs a value carried by the corresponding output wire. The operations considered for gates in this paper include in, ref, out, $\mathsf{reg}, \wedge, \vee, \neg, \overline{\wedge}, \overline{\vee}, \oplus, \overline{\oplus}$. Detailed explanations of these operations will be covered later in this section.

The physical circuits can be modeled as a masked circuit C [KSM20]. It is defined as a binary tuple $(\boldsymbol{x}, G)$, where $\boldsymbol{x} \in \mathrm{GF}_2^t$ is the set of secret variables and G is a labeled directed acyclic graph. However, the physical circuit typically does not take $\boldsymbol{x}$ as inputs. Instead, each secret variable $x_i \in \boldsymbol{x}$ is split into $d+1$ shares by secret sharing, i.e., $x_i = \bigoplus_{j=1}^{d+1} x_{ij}$ and fed to the circuit C. The set of shares is denoted by $Sh(\boldsymbol{x}) = \{x_{ij} | x_i = \bigoplus_{j=1}^{d+1} x_{ij}, i \in [1,t], j \in [1,d+1]\}$.

The elements and internal connections in physical circuits are modeled by the second element of the tuple, $G = (\mathcal{N}, \mathcal{E}, \boldsymbol{z}, \mathrm{op}, f)$. $\mathcal{N}$ is the set of vertices, where a vertex n is in $\mathcal{N}$ if and only if there exists a gate in the physical circuit mapped to n (we use vertex n and gate n interchangeably in this paper). The set $\mathcal{E}$ is the set of directed edges, where an edge $e = (n_j, n_i)$ is in $\mathcal{E}$ if and only if there is a wire connecting the input of gate n_j to the output of gate n_i in the physical circuit. In this case, n_i is considered a child of n_j. A unary gate n has only one child, denoted by $n.\mathrm{lft}$, while a binary gate n has two children, $n.\mathrm{lft}$ and $n.\mathrm{rgt}$.

The set of input variables of the circuit, denoted by $\boldsymbol{z}$, includes the set of shares $Sh(\boldsymbol{x})$, which is a uniform sharing of $\boldsymbol{x}$, and the set of fresh masks $\boldsymbol{r}$. These fresh masks are *independently, identically, and uniformly distributed* random Boolean variables.

The function *op* maps the vertex $n \in \mathcal{N}$ to its operation. Each in gate stores a certain share of a secret variable, and each ref gate stores a fresh mask in $\boldsymbol{r}$. A reg gate denotes a state-holding register and therefore a synchronization point. In contrast, an out gate denotes a pure output port: it forwards the value of its child and does not store state. A registered circuit output is represented explicitly by a reg gate followed by an out gate. Other gates perform regular Boolean operations, and their functionality is self-explanatory from their names. As a side note, unary gates are reg, out, and $\neg$, while binary gates include $\wedge$, $\vee$, $\overline{\wedge}$, $\overline{\vee}$, $\oplus$, and $\overline{\oplus}$. The in and ref gates do not necessarily have inputs because the values they output are modeled to have the previously described uniform distributions (or uniform sharing) in the characterization of circuit inputs $\boldsymbol{z}$.

Revision note on output-node semantics. An earlier version of this source treated out as an output register and consequently used it as a stopping point in the glitch-extended observation recursion. This revision separates the two meanings: reg is the only synchronization node, whereas out is a transparent output port. Accordingly, the glitch-extended observation set now recurses through out, and the probing-position set distinguishes register inputs from circuit output ports. The Boolean observation function remains $f_n = f_{n.\mathrm{lft}}$ for both unary node types.

The function $f : \mathcal{N} \to (\boldsymbol{z} \to \mathrm{GF}_2)$ maps the vertex $n \in \mathcal{N}$ in the graph to the Boolean function $f_n : \boldsymbol{z} \to \mathrm{GF}_2$ computed by the corresponding gate. This function is also referred to as the observation function. The specific definition of the observation function f_n of

gate n is as follows:

$$f_n = \begin{cases} x_{ij} & \mathrm{op}(n) = \mathsf{in}, \text{gate } n \text{ stores } x_{ij} \in Sh(\boldsymbol{x}) \\ r & \mathrm{op}(n) = \mathsf{ref}, \text{gate } n \text{ stores } r \in \boldsymbol{r} \\ f_{n.\mathrm{lft}} & \mathrm{op}(n) \in \{\mathsf{reg}, \mathsf{out}\} \\ \neg f_{n.\mathrm{lft}} & \mathrm{op}(n) = \neg \\ f_{n.\mathrm{lft}} \circ f_{n.\mathrm{rgt}}, & \circ = \mathrm{op}(n) \in \{\wedge, \vee, \overline{\wedge}, \overline{\vee}, \oplus, \overline{\oplus}\} \end{cases} . \quad (1)$$

Example 1. The algebraic expressions are shown in Figure 1b for the first order DOM scheme of multiplication over GF_2, i.e., $c = g(a, b) = ab$. In order not to leak the actual value of a, b and c in a first-order probing attack, secret inputs a and b are both split into two shares a_1, a_2, b_1, b_2 and a bit fresh mask r is introduced. To prevent glitches from propagating back to inputs, four registers x_1, x_2, x_3, x_4 are used to store the results of combinational logic g_1, g_2, g_3, g_4. A possible hardware implementation (an abstracted version ignoring the control or clock signals) for this scheme is shown in Figure 1a. The annotation i_j of an in gate indicates it stores the j-th share of secret variable x_i. The corresponding labeled directed acyclic graph of this implementation is shown in Figure 1c.

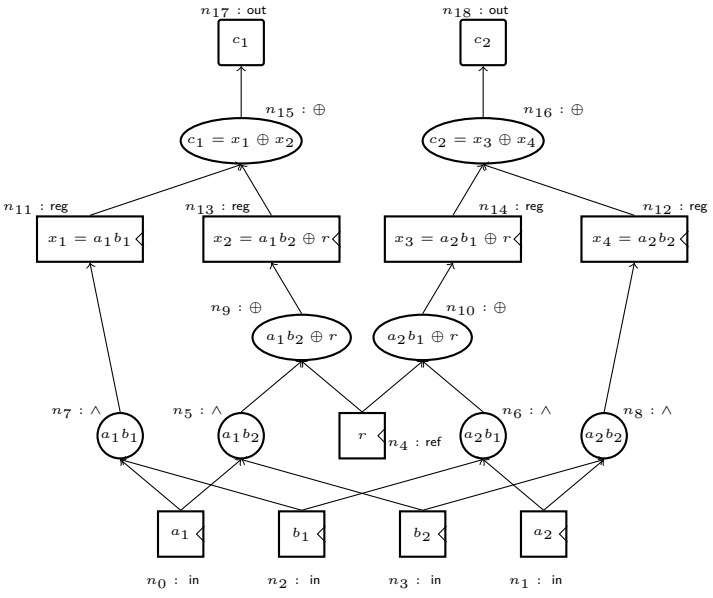
Input: a_1, a_2, b_1, b_2, r
Output: $c_1 = f_{n_{17}}, c_2 = f_{n_{18}}$

1: $f_{n_0} = \mathsf{in}\ 0_0\ a_1$
2: $f_{n_1} = \mathsf{in}\ 0_1\ a_2$
3: $f_{n_2} = \mathsf{in}\ 1_0\ b_1$
4: $f_{n_3} = \mathsf{in}\ 1_1\ b_2$
5: $f_{n_4} = \mathsf{ref}\ r$
6: $f_{n_5} = f_{n_0} \wedge f_{n_3}$
7: $f_{n_6} = f_{n_1} \wedge f_{n_2}$
8: $f_{n_7} = f_{n_0} \wedge f_{n_2}$
9: $f_{n_8} = f_{n_1} \wedge f_{n_3}$
10: $f_{n_9} = f_{n_5} \oplus f_{n_4}$
11: $f_{n_{10}} = f_{n_6} \oplus f_{n_4}$
12: $f_{n_{11}} = \mathsf{reg}\ f_{n_7}$
13: $f_{n_{12}} = \mathsf{reg}\ f_{n_8}$
14: $f_{n_{13}} = \mathsf{reg}\ f_{n_9}$
15: $f_{n_{14}} = \mathsf{reg}\ f_{n_{10}}$
16: $f_{n_{15}} = f_{n_{13}} \oplus f_{n_{11}}$
17: $f_{n_{16}} = f_{n_{14}} \oplus f_{n_{12}}$
18: $f_{n_{17}} = \mathsf{out}\ f_{n_{15}}$
19: $f_{n_{18}} = \mathsf{out}\ f_{n_{16}}$

(a) First order DOM implementation of multipilication

$g_1(a_1, b_1)$	$=$	a_1b_1	$\to x_1$	
$g_2(a_1, b_2, r)$	$=$	$a_1b_2 + r$	$\to x_2$	$x_1 + x_2 = c_1$
$g_3(a_2, b_1, r)$	$=$	$a_2b_1 + r$	$\to x_3$	$x_3 + x_4 = c_2$
$g_4(a_2, b_2)$	$=$	a_2b_2	$\to x_4$	

(b) First order DOM scheme of multipilication



(c) Graph representation of 1st order DOM implementation of multipilication

Figure 1: Example of masked hardware implementation

2.4 Security Model

The standard probing model [ISW03] and the glitch-extended probing model [FGP+18] are commonly used models for security analysis in software and hardware scenarios, respectively. In the standard probing model, an attacker can place a standard probe on a wire e to observe the value it carries. In contrast, in the glitch-extended probing model, a glitch-extended probe on a wire e enables the attacker to recover all the stable signals contributing to the value carried by e. For example, a standard probe on the output wire of gate n_{15} allows the attacker to recover the value of $f_{n_{15}}$, while a glitch-extended probe recovers the outputs of gates n_{11} and n_{13}, i.e., $\{f_{n_{11}}, f_{n_{13}}\}$.

We define an observation set O_n as the set of gates whose outputs are recovered by the attacker when placing a standard or glitch-extended probe on the output wire of gate n. The attacker can obtain the joint probability distribution of the observation function of the gates in the observation set. The calculation of observation set of a gate n under the standard probing model (*standard observation set*) is straightforward, i.e., $O_n = \{n\}$, while O_n in the glitch-extended probing model (*glitch-extended observation set*) is calculated as follows:

$$O_n = \begin{cases} \{n\}, & \text{if } \mathrm{op}(n) \in \{\mathsf{in}, \mathsf{ref}, \mathsf{reg}\} \\ O_{n.\mathrm{lft}}, & \text{if } \mathrm{op}(n) \in \{\neg, \mathsf{out}\} \\ O_{n.\mathrm{lft}} \cup O_{n.\mathrm{rgt}}, & \textit{otherwise} \end{cases}$$

In the standard (or glitch-extended) probing model with order d, an attacker can employ d standard (or glitch-extended) probes to interrogate the output wires of d gates in the masked circuit. Consequently, they can ascertain the joint probability distribution of observation functions in the union set of these d observation sets. Denoting a set of gates N where $|N| = d$, we represent $\mathcal{O}_d$ as the d-th order observation set, defined as $\mathcal{O}_d = \bigcup_{n_i \in N} O_{n_i}$.

The collection of observation functions corresponding to the gates in the observation set $\mathcal{O}$ is symbolized by $\boldsymbol{f}$, where $\boldsymbol{f} = \{f_n | n \in \mathcal{O}\}$. Each element in $\boldsymbol{f}$ is a single-output boolean function, collectively forming multi-output boolean functions defined over variables in $\boldsymbol{z}$.

Given a masked circuit $C = (\boldsymbol{x}, G)$ with $\boldsymbol{x}$ as its secret inputs, this paper defines the notion of standard and glitch-extended probing security [DBR19] as follows.

Definition 4 (d**-th Order Standard Probing Security**)**.** C is d-th order standard probing secure if for any set $N \subseteq \mathcal{N}$ with $|N| \leq d$, the *outputs* of the observation function set $\boldsymbol{f}$ corresponding to the standard observation set $\mathcal{O} = \bigcup_{n \in N} O_n$ is statistically independent of $\boldsymbol{x}$.

Definition 5 (d**-th Order Glitch-Extended Probing Security**)**.** C is d-th order glitch-extended probing secure if for any set $N \subseteq \mathcal{N}$ with $|N| \leq d$, the *outputs* of the observation function set $\boldsymbol{f}$ corresponding to the glitch-extended observation set $\mathcal{O} = \bigcup_{n \in N} O_n$ is statistically independent of $\boldsymbol{x}$.

It is important to note that $\mathcal{O}_d$ constitutes a set of gates, while $\boldsymbol{f}$ represents a set of functions, and statistical independence exists between sets of variables rather than between functions and variables. Thus, it is emphasized that the statistical independence in the above definition pertains to the *outputs of* $\boldsymbol{f}$ (which can be regarded as a set of variables) and $\boldsymbol{x}$. However, for brevity, less stringent terminology such as statistical independence between $\boldsymbol{f}$ and $\boldsymbol{x}$ or between $\mathcal{O}_d$ and $\boldsymbol{x}$ will be employed.

2.5 Statistical Independence Check Based on ROBDDs

SILVER[1], developed by the authors of [KSM20], stands as a prominent tool in the formal verification of masked implementations.

In practice, it is not necessary to verify the security of every d'-th ($d' \leq d$) order observation set $\mathcal{O}_{d'}$ to establish the security of a masked circuit. SILVER, depending on the selected model (standard or glitch-extended probing model), initially computes the set of probing positions $\mathcal{P}$. The security verification then focuses on all the d'-tuples ($d' \leq d$) from $\mathcal{P}$. The rationale for this simplification is as follows.

Since registers and output ports carry the same Boolean function as their left children, they are redundant probing positions under the standard probing model. Therefore they are excluded from $\mathcal{P}$. In other words, under the standard probing model, the set of probing positions is defined as $\mathcal{P} := \{n | \mathrm{op}(n) \in \{\mathsf{in}, \mathsf{ref}, \neg, \wedge, \vee, \overline{\wedge}, \overline{\vee}, \oplus, \overline{\oplus}\}\}$.

In the glitch-extended probing model, the relevant internal probing positions are the inputs of registers, because each such position dominates the combinational logic feeding that synchronization point. Circuit outputs must also be retained as probing positions when they are not followed by registers. Since out is a transparent port, probing an output node n yields $O_n = O_{n.\mathrm{lft}}$. Hence we use

$$\mathcal{P} := \{n.\mathrm{lft} \mid \mathrm{op}(n) = \mathsf{reg}\} \cup \{n \mid \mathrm{op}(n) = \mathsf{out}\}.$$

The number of probes is counted in the original probing-position set, whereas the statistical-independence check is performed on the union of the expanded observation sets. In particular, one output probe still counts as one probe even if its glitch-extended observation set contains several stable signals.

Given a masked implementation $C = (\boldsymbol{x}, G)$, SILVER initiates verification with $d = 1$. It examines the statistical independence between the outputs of the observation function set $\boldsymbol{f}$ and the set of secret variables $\boldsymbol{x}$ for each d-th order observation set $\mathcal{O}_d = \bigcup_{n \in R} O_n$, where $R \subseteq \mathcal{P}$ and $|R| = d$. If all $\mathcal{O}_d$ sets maintain security for the current d, SILVER increments d by 1 and continue to verify the security of all $\mathcal{O}_d$ sets with the incremented d. If any observation set $\mathcal{O}_d$ lacks statistical independence from $\boldsymbol{x}$, SILVER returns the actual security order $d - 1$ and the insecure probing-position set R.

The most critical and time-consuming step in the verification process involves confirming the statistical independence between $\boldsymbol{f}$ and $\boldsymbol{x}$. Since $\boldsymbol{f}$ and $\boldsymbol{x}$ are considered as two sets of boolean variables, Theorem 1 is introduced in [KSM20] to verify the independence between $\boldsymbol{f}$ and $\boldsymbol{x}$. Specifically, this theorem requires verifying the mutual independence of the event $\boldsymbol{f'} = \boldsymbol{\alpha}$ and the event $\boldsymbol{x'} = \boldsymbol{\beta}$ for each subset $\boldsymbol{f'}$ of $\boldsymbol{f}$ and each subset $\boldsymbol{x'}$ of $\boldsymbol{x}$.

Theorem 1. [KSM20] *Two sets of random boolean variables $\boldsymbol{x}, \boldsymbol{y}$ are statistically independent if and only if for all $\boldsymbol{x'} \subseteq \boldsymbol{x}, \boldsymbol{y'} \subseteq \boldsymbol{y}$, where $\boldsymbol{x'} \neq \emptyset$ and $\boldsymbol{y'} \neq \emptyset$, there exist $\boldsymbol{\alpha}, \boldsymbol{\beta}$ such that $p_{\boldsymbol{x'},\boldsymbol{y'}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = p_{\boldsymbol{x'}}(\boldsymbol{\alpha}) p_{\boldsymbol{y'}}(\boldsymbol{\beta})$, where $\boldsymbol{\alpha}, \boldsymbol{\beta}$ can be any two sets of boolean values.*

The verification of mutual independence relies on the equation $\Pr[\boldsymbol{f'} = \boldsymbol{\alpha}, \boldsymbol{x'} = \boldsymbol{\beta}] = \Pr[\boldsymbol{f'} = \boldsymbol{\alpha}] \Pr[\boldsymbol{x'} = \boldsymbol{\beta}]$. Note that each secret variable x_i in $\boldsymbol{x}$ is also perceived as boolean functions (defined over variables in $Sh(\boldsymbol{x})$). Consequently, this equation involves computing the joint probability of outputs of three multi-output boolean functions $\boldsymbol{f'} \cup \boldsymbol{x'}, \boldsymbol{f'}, \boldsymbol{x'}$ respectively, which is non-trivial. However, if $\boldsymbol{\alpha}$ is fixed to $2^{|\boldsymbol{f'}|} - 1$ (recall that an integer could be interpreted as a set of boolean values, mentioned in the Table 1), i.e., all elements in $\boldsymbol{f'}$ are assigned the value 1, the joint probability of $\boldsymbol{f'} = 2^{|\boldsymbol{f'}|} - 1$ equals $\Pr[\mathsf{f} = 1]$, where $\mathsf{f} := \bigwedge_{f_i \in \boldsymbol{f'}} f_i$ is a single-output boolean function. Similar computations apply for the probabilities $\Pr[\boldsymbol{x'} = 2^{|\boldsymbol{x'}|} - 1]$ and $\Pr[\boldsymbol{f'} = 2^{|\boldsymbol{f'}|} - 1, \boldsymbol{x'} = 2^{|\boldsymbol{x'}|} - 1]$. Fortunately, the output probability of a single-output boolean function can be efficiently computed using Reduced Ordered Binary Decision Diagrams (ROBDDs) [TN95, Mil98]. Hence, Theorem

[1] https://github.com/Chair-for-Security-Engineering/SILVER

1 can be utilized to prove statistical independence between $\boldsymbol{f'}$ and $\boldsymbol{x}$ through ROBDDs, with reductions from multi-output boolean functions to single-output boolean functions. Notably, according to [Mil98], the joint output probability of $\boldsymbol{f'} \cup \boldsymbol{x'}$ could be computed without constructing new ROBDDs.

To streamline representation, this paper introduces the product combination coefficient $\boldsymbol{\lambda} \in \mathrm{GF}_2^{|\boldsymbol{f}|}$ for each subset $\boldsymbol{f'}$ of $\boldsymbol{f}$. Specifically, $\forall f_i \in \boldsymbol{f}$, if $f_i \in \boldsymbol{f'}$, then $\lambda_i = 1$, otherwise $\lambda_i = 0$. The event $\boldsymbol{f'} = 2^{|\boldsymbol{f'}|} - 1$ can be denoted as $\bigwedge_{i=1}^{|\boldsymbol{f}|} f_i^{\lambda_i} = 1$, abbreviated as $\boldsymbol{f}^{\boldsymbol{\lambda}} = 1$. Thus, the probability equation $\Pr[\boldsymbol{f'} = \boldsymbol{\alpha}, \boldsymbol{x'} = \boldsymbol{\beta}] = \Pr[\boldsymbol{f'} = \boldsymbol{\alpha}] \Pr[\boldsymbol{x'} = \boldsymbol{\beta}]$ with $\boldsymbol{\alpha} = 2^{|\boldsymbol{f'}|} - 1$ and $\boldsymbol{\beta} = 2^{|\boldsymbol{x'}|} - 1$ can be expressed as

$$\Pr[\boldsymbol{f}^{\boldsymbol{\lambda}} = 1, \boldsymbol{x}^{\boldsymbol{\gamma}} = 1] = \Pr[\boldsymbol{f}^{\boldsymbol{\lambda}} = 1] \Pr[\boldsymbol{x}^{\boldsymbol{\gamma}} = 1] \tag{2}$$

Using this notation, $\boldsymbol{f}$ and $\boldsymbol{x}$ are statistically independent if and only if for all $0 \neq \boldsymbol{\lambda} \in \mathrm{GF}_2^{|\boldsymbol{f}|}, 0 \neq \boldsymbol{\gamma} \in \mathrm{GF}_2^{|\boldsymbol{x}|}$, Equation 2 holds.

3 Reduction Rules

SILVER encounters efficiency challenges, particularly when verifying larger implementations such as the masked S-box of AES. A significant contributing factor to this inefficiency is its exponential complexity. When verifying the statistical independence between the observation function set $\boldsymbol{f}$ and the secret variable set $\boldsymbol{x}$, SILVER needs to verify Equation 2 for $(2^{|\boldsymbol{x}|} - 1)(2^{|\boldsymbol{f}|} - 1)$ combinations. This complexity scales exponentially with the size of $\boldsymbol{f} \cup \boldsymbol{x}$.

To address this issue, we propose two Reduction Rules in this section to diminish the size of $\boldsymbol{f}$ and $\boldsymbol{x}$. Instead of constructing ROBDDs to precisely characterize the observation function f_n of a gate n, we utilize several data structures to store necessary information. This information is sufficient to infer that f_n has been masked by a fresh mask that is not used elsewhere (such a mask is referred to as a *perfect mask*). Consequently, the observation function f_n is *independently uniformly distributed*. Thus, it can be safely removed from the observation functions $\boldsymbol{f}$ without altering the result of the statistical independence check.

3.1 Auxiliary Data Structures

Inspired by prior work in formal verification of software masking [EWS14, OMHE17, ZGSW18], we introduce auxiliary data structures in this section to infer the independent uniform distribution of a single observation function.

Consider the example in Figure 1, where gate n_9 masks the results of n_5 ($a_1 b_2$) by adding a fresh mask r to it. Consequently, r ensures that the observation function $f_{n_9} = a_1 b_2 \oplus r$ follows a uniform distribution. We observe that if the expression of f_n can be rewritten as $r \oplus f'_n$, where r does not appear in the expression of f'_n (i.e., r is not a variable of f'_n), then f_n follows a uniform distribution and r is a perfect mask for f_n.

Note that r also satisfies the aforementioned property required to be a perfect mask of gates n_4, n_{13}, n_{15}, n_{17} and $n_{10}, n_{14}, n_{16}, n_{18}$. Essentially, the perfect masks of a non-leaf gate originate from its child gates, while a leaf gate uses, as a perfect mask, the independently uniformly distributed random variable it stores. In other words, r is transmitted as a perfect mask through two paths: $n_4 \to n_9 \to n_{13} \to n_{15} \to n_{17}$ and $n_4 \to n_{10} \to n_{14} \to n_{16} \to n_{18}$. This transmission requires two conditions. First, the parent gate n must be one of the gates that operates a function that is bijective to its inputs, i.e., binary gates $\{\oplus, \overline{\oplus}\}$ and unary gates $\{\neg, \mathsf{reg}, \mathsf{in}, \mathsf{out}\}$. Second, if one child gate have perfect masks that do not occur in the expression of the other child gate (if the other child exists), then these perfect masks are transmitted to the parent gate. Note

that the second condition implies that the transmitted perfect masks only occur once in the expression of the parent gate. Thus, f_n can be rewritten as $f'_n \oplus \bigoplus_{r\in \mathrm{perf}(n)} r$ where $\mathrm{perf}(n)$ denotes the set of perfect masks of gate n.

Now we introduce the method to compute $\mathrm{perf}(n)$ for every gate in C. We start with the leaf gates. Since each secret variable $x_i \in \boldsymbol{x}$ has $d+1$ shares and any selection of d shares of x_i are independent of each other and follows a uniform distribution [Bil15], the first d shares of x_i can function as perfect masks and the last share is seen as a function defined over variables in $\{x_i, x_{i1}, \cdots, x_{id}\}$, i.e., $x_{i,d+1} = x_i \oplus \bigoplus_{j=1}^{d} x_{ij}$. Following this observation, we have that for $1 \le i \le |\boldsymbol{x}|, 1 \le j \le d$, an in gate that store x_{ij} have perfect masks x_{ij} and for $1 \le i \le |\boldsymbol{x}|$, an in gate that store $x_{i,d+1}$ have perfect masks $\{x_{i1}, \cdots, x_{id}\}$. The perfect mask of ref gates is the fresh mask it stores.

It is not straightforward to obtain the perfect masks of a non-leaf gate n. First, we should obtain $\mathrm{supp}(n)$, the set of variables appearing in the expression of f_n. It is computed as follows:

$$\mathrm{supp}(n) = \begin{cases} \{f_n\}, & \text{if } \mathrm{op}(n) \in \{\mathsf{in}, \mathsf{ref}\}, f_n \neq x_{i,d+1} \\ \{x_i, x_{i1}, \ldots, x_{id}\}, & \text{if } \mathrm{op}(n) = \mathsf{in}, f_n = x_{i,d+1} \\ \mathrm{supp}(n.\mathrm{lft}), & \text{if } \mathrm{op}(n) \in \{\neg, \mathsf{reg}, \mathsf{out}\} \\ \mathrm{supp}(n.\mathrm{lft}) \cup \mathrm{supp}(n.\mathrm{rgt}), & \text{otherwise} \end{cases} \tag{3}$$

With $\mathrm{supp}(n)$, $\mathrm{perf}(n)$ can be computed using Equation 4. Note that in the second line of Equation 4, the left child $n.\mathrm{lft}$ transmits the subset of its perfect masks $\mathrm{perf}(n.\mathrm{lft}) \setminus \mathrm{supp}(n.\mathrm{rgt})$ to its parent. The elements in $\mathrm{perf}(n.\mathrm{lft}) \setminus \mathrm{supp}(n.\mathrm{rgt})$ are the perfect masks of $n.\mathrm{lft}$ that do not appear in the expression of $n.\mathrm{rgt}$, indicating that these masks will appear in the expression of n exactly once. The same analysis holds for $\mathrm{perf}(n.\mathrm{rgt}) \setminus \mathrm{supp}(n.\mathrm{lft})$. Equation 4 also ensures that the transmission of perfect masks only occurs between the aforementioned bijective gates (rather than non-bijective gates) and their children.

$$\mathrm{perf}(n) = \begin{cases} \mathrm{supp}(n) \setminus \boldsymbol{x}, & \text{if } \mathrm{op}(n) \in \{\mathsf{in}, \mathsf{ref}\} \\ (\mathrm{perf}(n.\mathrm{lft}) \setminus \mathrm{supp}(n.\mathrm{rgt})) \bigcup \\ \quad (\mathrm{perf}(n.\mathrm{rgt}) \setminus \mathrm{supp}(n.\mathrm{lft})), & \text{if } \mathrm{op}(n) \in \{\oplus, \overline{\oplus}\} \\ \mathrm{perf}(n.\mathrm{lft}), & \text{if } \mathrm{op}(n) \in \{\neg, \mathsf{reg}, \mathsf{out}\} \\ \emptyset, & \text{otherwise} \end{cases} \tag{4}$$

In addition to $\mathrm{supp}(n)$ and $\mathrm{perf}(n)$, we also define $\mathrm{nisupp}(n)$ to count the number of shares that appear in the expression of a gate. It can be computed using Equation 5. The difference between $\mathrm{supp}(n)$ and $\mathrm{nisupp}(n)$ is that the former considers $\boldsymbol{z} \cup \boldsymbol{x} \setminus \{x_{i,d+1} | 1 \le i \le |\boldsymbol{x}|\}$ as the set of input variables to the observation functions, while the latter considers only $\boldsymbol{z}$ as the set of input variables.

$$\mathrm{nisupp}(n) = \begin{cases} \{f_n\}, & \text{if } \mathrm{op}(n) \in \{\mathsf{in}, \mathsf{ref}\} \\ \mathrm{nisupp}(n.\mathrm{lft}), & \text{if } \mathrm{op}(n) \in \{\neg, \mathsf{reg}, \mathsf{out}\} \\ \mathrm{nisupp}(n.\mathrm{lft}) \cup \mathrm{nisupp}(n.\mathrm{rgt}), & \text{otherwise} \end{cases} \tag{5}$$

The auxiliary data structures are also defined over a set of gates N, namely $\mathrm{supp}(N) := \bigcup_{n\in N} \mathrm{supp}(n)$, $\mathrm{perf}(N) := \bigcup_{n\in N} \mathrm{perf}(n)$ and $\mathrm{nisupp}(N) := \bigcup_{n\in N} \mathrm{nisupp}(n)$.

3.2 Reduction Rules

From the last subsection, we understand that if the perfect mask set of a gate n is not empty, then the single observation function f_n has a uniform distribution. However, the observation set $\mathcal{O}_d$ usually contains more than one observation function. Even if each gate

in $\mathcal{O}_d$ has a non-empty perfect mask set, it is not sufficient to ensure that the observation functions $\boldsymbol{f}$ are jointly uniform. Consider three observation functions $f_{n_1} = a_1 + r_1$, $f_{n_2} = a_2 + r_2$, $f_{n_3} = r_1 + r_2$, where the secret variable a is split into two shares a_1 and a_2, and r_1 and r_2 are two fresh masks. Although n_1, n_2, and n_3 all have at least one perfect mask and each follows a uniform distribution, combining them will leak information about a. This is because the three functions share some identical perfect masks.

Given this observation, to establish that $\mathcal{O}_d$ follows a joint uniform distribution, a straightforward approach would involve ensuring that each gate $n \in \mathcal{O}_d$ possesses at least one perfect mask not utilized by other gates as support variables. In essence, for $1 \le i \le |\mathcal{O}_d|$, it is necessary that $n_i \in \mathcal{O}_d$ and $\mathrm{perf}(n_i) \setminus \bigcup_{j \ne i} \mathrm{supp}(n_j) \ne \emptyset$.

However, we demonstrate that proving the joint uniform distribution of $\mathcal{O}_d$ can be achieved through a less stringent condition by utilizing the following Reduction Rule.

Reduction Rule 1. *Given a d-th order observation set $\mathcal{O}_d$, if $\exists n_i \in \mathcal{O}_d$ (where i is the index of n_i in $\mathcal{O}_d$) such that $r \in perf(n_i)$ and $r \notin supp(\mathcal{O}_d \setminus \{n_i\})$, then $\mathcal{O}_d$ is statistically independent of $\boldsymbol{x}$ if and only if $\mathcal{O}_d \setminus \{n_i\}$ is statistically independent of $\boldsymbol{x}$.*

Proof. Let $f'_{n_i} = r + f_{n_i}$ be the de-masked observation function (then $r \notin \mathrm{supp}(f'_{n_i})$) and $\boldsymbol{f}_{\bar{i}} = \boldsymbol{f} \setminus \{f_{n_i}\}$ be the observation functions in $\mathcal{O}_d \setminus \{n_i\}$.

$\Longrightarrow$ We have $p_{\boldsymbol{f}_{\bar{i}},\boldsymbol{x}}(\boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{\beta}) = \sum_{\alpha_i} p_{\boldsymbol{f}_{\bar{i}},\boldsymbol{x},f_i}(\boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{\beta}, \alpha_i) = p_{\boldsymbol{x}}(\boldsymbol{\beta}) \sum_{\alpha_i} p_{\boldsymbol{f}_{\bar{i}},f_i}(\boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{\alpha}_i) = p_{\boldsymbol{f}_{\bar{i}}}(\boldsymbol{\alpha}_{\bar{i}}) \cdot p_{\boldsymbol{x}}(\boldsymbol{\beta})$. Hence, $\boldsymbol{f}_{\bar{i}}$, i.e., $\mathcal{O}_d \setminus \{n_i\}$ is statistically independent of $\boldsymbol{x}$.

$\Longleftarrow$ First, we show that f_{n_i} is statistically independent of $\boldsymbol{f}_{\bar{i}} \cup \boldsymbol{x}$.

$$
\begin{aligned}
&\Pr[\boldsymbol{f} = \boldsymbol{\alpha}, \boldsymbol{x} = \boldsymbol{\beta}] \\
&= \Pr[r + f'_{n_i} = \alpha_i, \boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] \\
&= \Pr[r = 0, f'_{n_i} = \alpha_i, \boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] + \Pr[r = 1, f'_{n_i} = \neg\alpha_i, \boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] \\
&= \frac{1}{2}\Pr[f'_{n_i} = \alpha_i, \boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] + \frac{1}{2}\Pr[f'_{n_i} = \neg\alpha_i, \boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] \\
&= \frac{1}{2}\Pr[\boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] \\
&= \Pr[f_{n_i} = \alpha_i]\Pr[\boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}]
\end{aligned}
$$

The fourth line of the equation holds because r does not appear in the support variable set of $\{f'_{n_i}\} \cup \boldsymbol{f}_{\bar{i}} \cup \boldsymbol{x}$. The fifth line holds due to the law of total probability.

Similarly, we can prove that f_{n_i} is also statistically independent of $\boldsymbol{f}_{\bar{i}}$.

Therefore $\Pr[\boldsymbol{f} = \boldsymbol{\alpha}, \boldsymbol{x} = \boldsymbol{\beta}] = \Pr[f_{n_i} = \alpha_i]\Pr[\boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}, \boldsymbol{x} = \boldsymbol{\beta}] = \Pr[f_{n_i} = \alpha_i]\Pr[\boldsymbol{f}_{\bar{i}} = \boldsymbol{\alpha}_{\bar{i}}]\Pr[\boldsymbol{x} = \boldsymbol{\beta}] = \Pr[\boldsymbol{f} = \boldsymbol{\alpha}]\Pr[\boldsymbol{x} = \boldsymbol{\beta}]$.

We complete the proof. □

If for $1 \le i \le |\mathcal{O}_d|$, it holds that $n_i \in \mathcal{O}_d$ and $\mathrm{perf}(n_i) \setminus \bigcup_{j>i} \mathrm{supp}(n_j) \ne \emptyset$, we can apply Reduction Rule 1 for $|\mathcal{O}_d|$ times to conclude the security (or uniformity) of $\mathcal{O}_d$. This is a weaker condition than the straightforward method. Below is an example.

Example 2. Let the set of secret variables be $\{a\}$, the set of shares be $\{a_1, a_2\}$, the set of fresh masks be $\{r_1, r_2\}$, and the set of observation functions be $\boldsymbol{f} = \{f_{n_1} = r_1 + a_1 r_2, f_{n_2} = a_1 + r_2, f_{n_3} = a_2\}$. It is easy to obtain that $\mathrm{perf}(n_1) = \{r_1\}$, $\mathrm{perf}(n_2) = \{a_1, r_2\}$, and $\mathrm{perf}(n_3) = \{a_1\}$. First, since $r_1 \in \mathrm{perf}(n_1)$ and $r_1 \notin \mathrm{supp}(n_2, n_3)$, n_1 can be removed from $\boldsymbol{f}$, i.e., $\boldsymbol{f}$ is reduced to $\{f_{n_2}, f_{n_3}\}$. Next, with $r_2 \in \mathrm{perf}(n_2)$ and $r_2 \notin \mathrm{supp}(n_3)$, n_2 is removed. And in one more step, n_3 is also removed.

Note that Reduction Rule 1 aligns with the concept of OPT rule (or so-called optimistic sampling rule) of maskVerif [BBD+15, BBC+19]. maskVerif utilizes expression substitution to eliminate occurrences of secret variables. Consider two secret inputs a, b and their corresponding shares a_1, b_1, a_2, b_2 and a computation a_2b_2. maskVerif will regard a, b, a_1, b_1 as variables while a_2, b_2 are expressions defined over (a, b, a_1, b_1), i.e., $a_2 := a + a_1, b_2 := b+b_1$. So the computation a_2b_2 is expressed as $(a+a_1)(b+b_1)$. We can see that $a_2 = a+a_1$ utilizes a fresh mask a_1 (not used by b_2), ensuring that a_2 has the same distribution as a_1. Therefore, it is safe to substitute $a + a_1$ with a_1 in expression $(a + a_1)(b + b_1)$. After substitution, the distribution of $a_1(b + b_1)$ remains unchanged from the original expression. The same applies to substituting $b+b_1$ with b_1. Ultimately, the computation $(a+a_1)(b+b_1)$, which initially includes secret inputs a, b, is reduced to an expression a_1b_1 that contains no secret inputs. While maskVerif achieves this substitution using imperative graphs, we employ similar techniques through the data structures outlined in subsection 3.1. However, we could not conclude that a_2b_2 is secure through Reduction Rule 1 since it has no perfect masks. This also motivates the introduction of Reduction Rule 2.

A more apt example illustrating the difference between rule OPT and Reduction Rule 1 would involve verifying $a_2 + b_2$. Due to the bijection between a_1 (or b_1) and $a_2 + b_2$, maskVerif would simplify this expression to a_1 (or b_1), which contains no secret inputs, whereas Reduction Rule 1 identifies a_1 and b_1 as the perfect masks of $a_2 + b_2$, thereby concluding its security.

While Reduction Rule 1 reduces the size of $\mathcal{O}_d$, we now introduce a Reduction Rule which reduces the size of $\boldsymbol{x}$.

Let $\boldsymbol{x}_I$ be the set of secret variables all shares of which have been used by $\boldsymbol{f}$, i.e., $\boldsymbol{x}_I = \{x_i | Sh(x_i) \subseteq nisupp(\mathcal{O}_d), 1 \leq i \leq |\boldsymbol{x}|\}$ and $\boldsymbol{x}_{\bar{I}} = \boldsymbol{x} \setminus \boldsymbol{x}_I$. We call $\boldsymbol{x}_I$ probing-related variables, while variables in $\boldsymbol{x}_{\bar{I}}$ are probing-independent variables. Then Reduction Rule 2 holds.

Reduction Rule 2. *Given a d-th order observation set $\mathcal{O}_d$, $\mathcal{O}_d$ is statistically independent of $\boldsymbol{x}$ if and only if $\mathcal{O}_d$ is statistically independent of $\boldsymbol{x}_I$.*

Intuitively, $\mathcal{O}_d$ does not use all shares of $\boldsymbol{x}_{\bar{I}}$, so it is statistically independent of $\boldsymbol{x}_{\bar{I}}$. However it is not sufficient to prove the independence between $\mathcal{O}_d$ and $\boldsymbol{x}$ with $\mathcal{O}_d$ statistically independent of $\boldsymbol{x}_I$ and $\boldsymbol{x}_{\bar{I}}$ respectively.

Proof. Let I be the set of index of variables in $\boldsymbol{x}_I$ and $\bar{I}$ be $\{1, \cdots, |\boldsymbol{x}|\} \setminus I$.

$\Longrightarrow$ We have $p_{\boldsymbol{f},\boldsymbol{x}_I}(\boldsymbol{\alpha}, \boldsymbol{\beta}_I) = \sum_{\boldsymbol{\beta}_{\bar{I}}} p_{\boldsymbol{f},\boldsymbol{x}_I,\boldsymbol{x}_{\bar{I}}}(\boldsymbol{\alpha}, \boldsymbol{\beta}_I, \boldsymbol{\beta}_{\bar{I}}) = p_{\boldsymbol{f}}(\boldsymbol{\alpha})(\sum_{\boldsymbol{\beta}_{\bar{I}}} p_{\boldsymbol{x}_I,\boldsymbol{x}_{\bar{I}}}(\boldsymbol{\beta}_I, \boldsymbol{\beta}_{\bar{I}})) = p_{\boldsymbol{f}}(\boldsymbol{\alpha})\ p_{\boldsymbol{x}_I}(\boldsymbol{\beta}_I)$. Hence, $\mathcal{O}_d$ is statistically independent of $\boldsymbol{x}_I$.

$\Longleftarrow$ Let $\boldsymbol{s}_I$ be the set $\{x_{ij} | x_i \in \boldsymbol{x}_I, 1 \leq j \leq d\}$, and $\boldsymbol{s}_{\bar{I}}$ be the set $Sh(\boldsymbol{x}_{\bar{I}}) \cap nisupp(\mathcal{O}_d)$. Since for all $x_i \in \boldsymbol{x}_I$, the last share $x_{i,d+1}$ could be viewed as a function defined over $\{x_i, x_{i1}, \cdots, x_{id}\}$, i.e., $x_{i,d+1} = x_i \oplus \bigoplus_{i=1}^{d} x_{ij}$, then $\boldsymbol{f}$ can be seen as a multi-output boolean function defined over variables $\boldsymbol{x}_I \uplus \boldsymbol{s}_I \uplus \boldsymbol{s}_{\bar{I}} \uplus \boldsymbol{r}$, i.e., $\boldsymbol{f}(\boldsymbol{x}_I, \boldsymbol{s}_I, \boldsymbol{s}_{\bar{I}}, \boldsymbol{r})$.

Note that all variables in $\boldsymbol{s}_I \uplus \boldsymbol{s}_{\bar{I}} \uplus \boldsymbol{r}$ are independently distributed random variables, then the following holds.

$$
\begin{aligned}
& \boldsymbol{f} \text{ is statistically independent of } \boldsymbol{x}_I. \\
\Longleftrightarrow\ & \forall \boldsymbol{\alpha} \in \mathrm{GF}_2^{|\boldsymbol{f}|}, \nexists (\boldsymbol{\beta}^{(1)}, \boldsymbol{\beta}^{(2)}) \in (\mathrm{GF}_2^{|\boldsymbol{x}_I|})^2 \text{ satisfying } \boldsymbol{\beta}^{(1)} \neq \boldsymbol{\beta}^{(2)}, \\
& \text{such that } \Pr[\boldsymbol{f}(\boldsymbol{\beta}^{(1)}, \boldsymbol{s}_I, \boldsymbol{s}_{\bar{I}}, \boldsymbol{r}) = \boldsymbol{\alpha}] \neq \Pr[\boldsymbol{f}(\boldsymbol{\beta}^{(2)}, \boldsymbol{s}_I, \boldsymbol{s}_{\bar{I}}, \boldsymbol{r}) = \boldsymbol{\alpha}] \\
\Longrightarrow\ & \forall \boldsymbol{\alpha} \in \mathrm{GF}_2^{|\boldsymbol{f}|}, \nexists (\boldsymbol{\beta}^{(1)}, \boldsymbol{\beta}^{(2)}) \in (\mathrm{GF}_2^{|\boldsymbol{x}|})^2, \text{ satisfying } \boldsymbol{\beta}^{(1)} \neq \boldsymbol{\beta}^{(2)}, \\
& \text{such that } \Pr[\boldsymbol{f}(\boldsymbol{\beta}_I^{(1)}, \boldsymbol{s}_I, \boldsymbol{s}_{\bar{I}}, \boldsymbol{r}) = \boldsymbol{\alpha}] \neq \Pr[\boldsymbol{f}(\boldsymbol{\beta}_I^{(2)}, \boldsymbol{s}_I, \boldsymbol{s}_{\bar{I}}, \boldsymbol{r}) = \boldsymbol{\alpha}] \\
\Longleftrightarrow\ & \boldsymbol{f} \text{ is statistically independent of } \boldsymbol{x}.
\end{aligned}
$$

The implies can be easily verified by proof of contradiction.

□

Below is an example of applications of Reduction Rule 2. It should be noted this observation function set cannot be reduced by Reduction Rule 1.

Example 3. Let the secret variable set be $\{a, b, c\}$, the shares be $\{a_1, a_2, b_1, b_2, c_1, c_2\}$, and the observation function set be $\{f_{n_1} = a_2 b_1, f_{n_2} = a_2 c_2, f_{n_3} = b_1 c_2\}$. Then $\text{nisupp}(f) = \{a_2, b_1, c_2\}$. Only one share of a, b, c appears in $\boldsymbol{f}$. Since $\boldsymbol{x}_I$ is empty, this observation function set is secure according to the rule 2.

Reduction Rule 2 shares similarities with checking the non-interference property (NI) of an observation set, or verifying the non-completeness property in Threshold Implementations. However, NI mandates at most d shares of all secret variables occur in the expression, whereas Reduction Rule 2 allows the presence of secret variables whose every share appears in the observation set.

Algorithm 1 presents the reduction algorithm based on the above Reduction Rules.

The function ReduceF($\mathcal{O}_d$) implements Reduction Rule 1. It attempts to find a gate n that uses a fresh mask not used by other gates (line 3). Such gate n is then removed from $\mathcal{O}_d$. The resulting set $\mathcal{O}_d \setminus \{n\}$ is further reduced by rule 1 in line 4.

The function ReduceX($\mathcal{O}_d, \boldsymbol{x}$) implements the reduction of the secret variable set. It adds the secret variable, all shares of which have been used by $\mathcal{O}_d$, to $\boldsymbol{x}_I$ and returns the probing-related variables set $\boldsymbol{x}_I$.

Algorithm 1 Reduction Algorithm

```
1:  function ReduceF(O_d)                                   ▷ Reduction Rule 1
2:      for all n_i ∈ O_d do
3:          if perf(n_i) \ supp(O_d \ {n_i}) ≠ ∅ then
4:              return ReduceF(O_d \ {n_i})
5:          end if
6:      end for
7:      return O_d
8:  end function
9:  function ReduceX(O_d, x)                                ▷ Reduction Rule 2
10:     x_I ← ∅
11:     for all x ∈ x do
12:         if Sh(x) \ nisupp(O_d) = ∅ then
13:             x_I.add(x)
14:         end if
15:     end for
16:     return x_I
17: end function
```

4 The Verification Tool - Prover

In this section, we introduce the formal verification tool, Prover, which is a modification of SILVER designed to efficiently verify the security of masked implementations.

4.1 Variable Ordering

The complexity of computing $\Pr[\boldsymbol{f}^{\boldsymbol{\lambda}} = 1, \boldsymbol{x}^{\boldsymbol{\gamma}} = 1]$ is $O(N(\boldsymbol{f}^{\boldsymbol{\lambda}}) \cdot N(\boldsymbol{x}^{\boldsymbol{\gamma}}))$ [Mil98], where $N(f)$ denotes the number of nodes in the ROBDD representing the boolean function f.

Therefore the complexity of checking statistical independence between $\boldsymbol{f}$ and $\boldsymbol{x}$ will be $O((\sum_{\boldsymbol{\lambda}\neq 0} N(\boldsymbol{f}^{\boldsymbol{\lambda}})) \cdot (\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}})))$. SILVER does not impose any restrictions on variable ordering when constructing ROBDDs. However, optimizing the variable ordering can significantly reduce $\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}})$, thereby improving performance.

Specifically, SILVER determines variable ordering based on annotations in the internal netlist file (or internal representation of the circuit graph), which are provided by its Verilog parser. The opening lines of the internal netlist file specify the input variables of the circuit. Each of these lines declares an input variable, and SILVER assigns the i-th BDD variable to the input variable listed on the i-th line. For instance, in a 2-shared Verilog implementation of the AES S-box with secret inputs $\{a, b, \cdots, h\}$. The variables are typically declared in the following order in the internal representation file produced by the Verilog parser: $a_1 \prec b_1 \prec \cdots \prec h_1 \prec a_2 \prec \cdots \prec h_2$. SILVER assigns these input variables as BDD variables in exactly the same sequential order.

As discussed in [HSSW10], this ordering is worst for computing a boolean function of the form $(a_1 \oplus a_2) \wedge \cdots \wedge (h_1 \oplus h_2)$ (the same form as $\boldsymbol{x}^{\boldsymbol{\gamma}}$). When the hamming weight $HW(\boldsymbol{\gamma})$ of $\boldsymbol{\gamma}$ is u, the ROBDD representing $\boldsymbol{x}^{\boldsymbol{\gamma}}$ would contain $3 \cdot 2^u - 1$ nodes. Suppose there are m input variables, then $\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}}) = \sum_{u=1}^{m} \binom{m}{u}(3 \cdot 2^u - 1) = 3^{m+1} - 2^m - 2$.

It is established in [HSSW10] that an optimal ordering would be $a_1 \prec a_2 \prec b_1 \prec b_2 \prec \cdots \prec h_1 \prec h_2$. With this optimal ordering, when $HW(\boldsymbol{\gamma}) = u$, the number of nodes in the corresponding ROBDD will be linear in u, i.e., $3u + 2$. Using the optimal ordering reduces the size of $\boldsymbol{x}^{\boldsymbol{\gamma}}$ from exponential complexity to linear complexity in the hamming weight of $\boldsymbol{\gamma}$. And the term $\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}})$ is simplified to $\sum_{u=1}^{m} \binom{m}{u}(3u + 2) = (3m + 4)2^{m-1} - 2$, where m is the number of input variables.

However, there is another method to further reduce the size of the ROBDD representation of $\boldsymbol{x}^{\boldsymbol{\gamma}}$. The concept behind this approach is quite similar to how variables are treated as support variables in the computation of supp(n) and nisupp(n). Instead of declaring a_1 and a_2 as ROBDD variables and computing the secret variable a as $a_1 + a_2$, we could declare a and a_1 as ROBDD variables and compute a_2 as $a + a_1$. The same approach is applied to $b, \cdots, h$ and their respective shares. In this scenario, the variable ordering would be $a \prec b \prec \cdots \prec h \prec a_1 \prec b_1 \prec \cdots \prec h_1$. When $HW(\boldsymbol{\gamma}) = u$, the number of nodes in the corresponding ROBDD will be $u + 2$. The term $\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}})$ is simplified to $\sum_{u=1}^{m} \binom{m}{u}(u + 2) = (m + 4)2^{m-1} - 2$, which represents approximately one-third of the optimal case.

When m is eight, the term $\sum_{\boldsymbol{\gamma}\neq 0} N(\boldsymbol{x}^{\boldsymbol{\gamma}})$ under the first two variable ordering strategies will be 19425 and 3582, respectively, which is approximately 12.5 times and 2.3 times larger than the third case, which is 1552.

However, we can not determine whether $N(\boldsymbol{f}^{\boldsymbol{\lambda}})$ grows larger or smaller compared to the original ordering in SILVER, because $\boldsymbol{f}^{\boldsymbol{\lambda}}$ is not as straightforward as a boolean function like $\boldsymbol{x}^{\boldsymbol{\gamma}}$. Additionally, analyzing $\boldsymbol{x}^{\boldsymbol{\gamma}}$ becomes more complex when $x \in \boldsymbol{x}$ has more than two shares. In fact, improving the variable ordering of ROBDDs is NP-complete [BW96]. Therefore, our objective is not to find a universally optimal ordering that suits every implementation. Instead, we aim to identify variable orderings that prove efficient in practical scenarios. To this end, we conducted an experiment to compare SILVER's performance under different variable orderings, with detailed results presented in Section 5. The choice of variable ordering is configurable as a user option in Prover. As a side note, the variable ordering of random inputs follows after the secrets and their shares. However, we did not specifically optimize the ordering of these variables.

4.2 Verification of Standard and Glitch-Extended Probing Security

Now, let us delve into the verification algorithm for standard and glitch-extended probing security. The overall algorithm is outlined in Algorithm 2. The fundamental idea behind

Algorithm 2 Improved d-th order Standard and Glitch-Extended Probing Security Verification Algorithm based on Reduction Rules and ROBDDs

Input: $\boldsymbol{x}, Sh(\boldsymbol{x}), \boldsymbol{r}, \mathcal{N}, \mathcal{E}, \text{op}, f, b$
Output: observation set R which leaks information about $\boldsymbol{x}$

function VerifyInc($\boldsymbol{x}, Sh(\boldsymbol{x}), \boldsymbol{r}, \mathcal{N}, \mathcal{E}, \text{op}, f, b$)
  for all $n \in \mathcal{N}$ **do** $f_n \leftarrow$ ComputeBDDFunction(n) ▷ According to Equation 1
  end for
  $\mathcal{P} \leftarrow \{n.\text{lft}|\text{op}(n) = \mathsf{reg}\} \cup \{n|\text{op}(n) = \mathsf{out}\}$ ▷ Positions in glitch-extended probing model
  if b **then** $\mathcal{P} \leftarrow \{n|\text{op}(n) \in \{\mathsf{in}, \mathsf{ref}, \neg, \wedge, \vee, \overline{\wedge}, \overline{\vee}, \oplus, \overline{\oplus}\}\}$ ▷ Positions in standard probing model
  end if
  $d' \leftarrow 1$
  $\mathcal{V} \leftarrow \{\emptyset\}$
  while true do
    for all $R \subseteq \mathcal{P}$ with $|R| = d'$ and the size of $\mathcal{O}_{d'} = \bigcup_{n \in R} O_n$ in a descending order **do**
      $\mathcal{O}_{d'} \leftarrow$ ReduceF($\mathcal{O}_{d'}$) ▷ Reducing $\mathcal{O}_{d'}$ using Reduction Rule 1
      if b and $|\mathcal{O}_{d'}| \neq d'$ or $\mathcal{O}_{d'} = \emptyset$ **then continue**
      end if
      $\boldsymbol{x}_I \leftarrow$ ReduceX($\mathcal{O}_{d'}, \boldsymbol{x}$), $t \leftarrow |\boldsymbol{x}_I|$ ▷ Reducing $\boldsymbol{x}$ using Reduction Rule 2
      if $t = 0$ **then continue**
      end if
      for all $\mathcal{O} \in \mathcal{V}$ **do**
        if $\mathcal{O}_{d'} \subseteq \mathcal{O}$ **then continue**
        end if
      end for
      $\boldsymbol{f} \leftarrow \bigcup_{n \in \mathcal{O}_{d'}} f_n$, $p \leftarrow |\boldsymbol{f}|$
      for all $\boldsymbol{\gamma} \in [1, 2^t - 1]$ **do**
        $x_{\boldsymbol{\gamma}} \leftarrow 1$
        for all $1 \leq i \leq t$ **do**
          $x_{\boldsymbol{\gamma}} \leftarrow x_{\boldsymbol{\gamma}} \wedge x_i^{\gamma_i}$ ▷ x_i is the i-th element of $\boldsymbol{x}_I$
        end for
      end for
      for $\boldsymbol{\lambda} = 2^p - 1$ down to 1 **do**
        $f_{\boldsymbol{\lambda}} \leftarrow 1$
        for all $1 \leq i \leq p$ **do**
          $f_{\boldsymbol{\lambda}} \leftarrow f_{\boldsymbol{\lambda}} \wedge f_i^{\lambda_{p-i}}$
        end for
        for all $\boldsymbol{\gamma} \in [1, 2^t - 1]$ **do**
          if $\Pr[f_{\boldsymbol{\lambda}} = 1, x_{\boldsymbol{\gamma}} = 1] \neq \Pr[f_{\boldsymbol{\lambda}} = 1] \Pr[x_{\boldsymbol{\gamma}} = 1]$ **then**
            return R
          end if
        end for
        if b **then break** ▷ Optimization for verifying standard probing security
        end if
      end for
      $\mathcal{V}$.add($\mathcal{O}_{d'}$)
    end for
    $d' \leftarrow d' + 1$
  end while
end function

verifying both security notions is to initially reduce the size of $\mathcal{O}_d$ and $\boldsymbol{x}$, and then employ ROBDDs to check the statistical independence between the reduced $\mathcal{O}_d$ and $\boldsymbol{x}$.

This algorithm takes the circuit C and a boolean value b as inputs and outputs a set of gates R with leakage about the secret variables $\boldsymbol{x}$. As explained in section 2.3, the inputs of the circuit are $Sh(\boldsymbol{x})$ and $\boldsymbol{r}$, and the information about the gates is stored in $(\mathcal{N}, \mathcal{E}, \text{op}, f)$. If b is True, then the algorithm verifies standard probing security. Otherwise, it verifies glitch-extended probing security.

First, Prover computes the ROBDD representation of all the observation functions $\{f_n | n \in \mathcal{N}\}$ (line 2). Based on the value of b, i.e., the selected security model, it chooses the set of positions $\mathcal{P}$ to be verified in the circuit.

Then, the verification starts with $d = 1$ (line 7). First, Prover utilizes Reduction Rule 1 to reduce the size of $\mathcal{O}_d$ (line 11). If Reduction Rule 1 fails to prove the security of $\mathcal{O}_d$, Prover reduces the secret variable set (line 14). If there is no probing-related variable, then it is secure, and no further verification is needed (line 15). Note that it is better to apply Reduction Rule 2 after applying Reduction Rule 1 because after Reduction Rule 1 is applied, the size of the reduced $\mathcal{O}_d$ is much smaller and depends on fewer variables.

Notably, we employ a strategy that could reduce the number of times to check statistical independence by ROBDDs. During the verification, Prover stores the $\mathcal{O}_d$ that has been verified to be secure by ROBDD into a list $\mathcal{V}$. Whenever Reduction Rule 1 and 2 fails to verify $\mathcal{O}_d$ is secure, Prover compares it to the elements in $\mathcal{V}$. If the $\mathcal{O}_d$ under verification happens to be a subset of a verified secure observation set, then it is secure and needs no further verification. This avoids the time-consuming process by ROBDDs. To employ the subset strategy, we sort the observation set by size in descending order and start the verification with the larger sets. This strategy is implemented in lines 8, 10, 18, and 41 of Algorithm 2.

One might assume that storing verified secure observation sets requires substantial RAM usage. However, we store only the corresponding gate number n of verified functions instead of their BDD representations. Each observation function's gate number n is a unique 4-byte integer, and the observation set O_d exclusively comprises these gate numbers.

Finally, if Rule 2 fails to return an empty set, it is necessary to invoke ROBDDs to verify the probability equations (lines 21 to 40). If Equation 2 does not hold, then leakage is detected. In this case, Prover returns the set of gates whose input wires leak information about $\boldsymbol{x}$.

However, there is a key difference in the observation sets between the standard and glitch-extended probing model, which could lead to some optimizations (lines 12 and 38) in the verification of the standard probing security. Note that a d-th order observation set $\mathcal{O}_d$ under glitch-extended probing may have an arbitrary size greater than or equal to d, while $\mathcal{O}_d$ in the standard probing model is always of size d. Normally, $\mathcal{O}_d$ needs to be reduced to an empty set to avoid the statistical independence check via ROBDDs. However, since d' increases from 1 in Algorithm 2, all sets of size smaller than the current d' have been verified in the previous iteration when verifying standard probing security. As a result, when the size of $\mathcal{O}_{d'}$ is reduced to a size less than d', the remaining set has already been checked in the previous iteration and needs no further verification (line 12 in Algorithm 2). If after reduction, the size of $\mathcal{O}_{d'}$ is still d', then ROBDDs are expected to be used to check whether Equation 2 holds for all $\boldsymbol{\lambda} \in [1, 2^d - 1]$. Again, for $\boldsymbol{\lambda}$ s.t. $HW(\boldsymbol{\lambda}) < d$, the independence checking has already been performed or implied by Reduction Rules in previous iterations. Thus, Prover only needs to check whether Equation 2 holds when $HW(\boldsymbol{\lambda}) = d'$ (line 38).

4.3 Verification of Uniformity

Uniformity (or uniform sharing) is an important property to maintain in Threshold Implementation.

Definition 6 (**Uniform Sharing**[KSM20])**.** Let $\boldsymbol{y}$ be a set of binary random variable and $Sh(\boldsymbol{y})$ its corresponding Boolean sharing. Each variable $y_i \in \boldsymbol{y}$ is split into $d+1$ shares y_{ij} $(1 \leq j \leq d+1)$. Then $Sh(\boldsymbol{y})$ is a uniform sharing of $\boldsymbol{y}$ iff $\forall \boldsymbol{\alpha}, \boldsymbol{\beta}$, Equation 6 holds.

$$\Pr[Sh(\boldsymbol{y}) = \boldsymbol{\alpha} | \boldsymbol{y} = \boldsymbol{\beta}] = \begin{cases} \frac{1}{2^{|\boldsymbol{y}|d}} & \text{if } \boldsymbol{\alpha} \text{ is a valid sharing for } \boldsymbol{\beta} \\ 0 & \text{else} \end{cases} \tag{6}$$

In the definition of uniformity, valid sharing means that for $1 \leq i \leq |\boldsymbol{y}|$, $\beta_i = \bigoplus_{j=1}^{d+1} \alpha_{ij}$ with $1 \leq j \leq d+1$.

In [KSM20], Lemma 1 (Lemma 4 in [KSM20]) was employed to verify the uniformity of the output sharing $Sh(\boldsymbol{y}) = \{y_{ij} | y_i = \bigoplus_{j=1}^{d+1} y_{ij}, 1 \leq i \leq |\boldsymbol{y}|, 1 \leq j \leq d+1\}$ for a Boolean function f with multiple outputs $\boldsymbol{y}$.

Lemma 1. *The output sharing $Sh(\boldsymbol{y})$ of a circuit C is uniform. $\iff$ Any selection of up to $|\boldsymbol{y}| \cdot d$ output shares is balanced excluding the cases where all $d+1$ shares of the same output are involved in the selection. $\iff$ $\Pr[\bigoplus_{i=1}^{|\boldsymbol{y}|} \boldsymbol{\lambda}^{(i)} \boldsymbol{y}_{\boldsymbol{s}}^{(i)} = 1] = \frac{1}{2}$ holds for all $\boldsymbol{\lambda}^{(i)} \in \mathrm{GF}_2^{d+1} (1 \leq i \leq |\boldsymbol{y}|)$ with $0 \leq HW(\boldsymbol{\lambda}^{(i)}) \leq d$ where $\boldsymbol{y}_{\boldsymbol{s}}^{(i)}$ denotes for $Sh(y_i)$ and $\boldsymbol{\lambda}^{(i)}$s are not all zeros.*

The second iff in Lemma 1 is not explicitly demonstrated in [KSM20]. However, it can be readily inferred from the following XOR Lemma in [Fri92], thus we omit its proof here.

Lemma 2 (XOR Lemma)**.** *A set of random variables $\boldsymbol{f} = \{f_0, f_1, \cdots, f_{n-1}\}$ follows an independent uniform distribution if and only if the following equation holds.*

$$\forall \boldsymbol{\lambda} \in [1, 2^n - 1], \Pr[\boldsymbol{\lambda} \boldsymbol{f} = 1] = \frac{1}{2} \tag{7}$$

The second iff in Lemma 1 constitutes the actual verification approach adopted by SILVER. $\boldsymbol{\lambda}^{(i)}$ selects at most d shares of y_i, and all shares selected by these $\boldsymbol{\lambda}^{(i)}$s are XORed to form a single-output function $f = \bigoplus_{i=1}^{|\boldsymbol{y}|} \boldsymbol{\lambda}^{(i)} \boldsymbol{y}_{\boldsymbol{s}}^{(i)}$. To verify the uniformity of a $(d+1)$-shared boolean function with an n-bit output, SILVER needs to construct $(2^{d+1}-1)^n - 1$ ROBDDs for such single-output function f and check whether f is balanced. For example, in the case of a 3-shared boolean function with 8-bit outputs, the number of times ROBDDs need to be constructed becomes $7^8 - 1 \approx 2^{22.5}$. Since the ROBDDs of the output functions of circuit C contain more nodes than the internal functions in C, the XOR operations on these ROBDDs become less efficient. Consequently, SILVER may not be able to verify the uniformity of several paired second-order masked S-boxes within a 24-hour time frame.

We now incorporate Reduction Rule 1 into the uniformity check to enhance its efficiency. As analyzed in Section 3.2, Reduction Rule 1 can be utilized to verify the uniformity of an observation set or reduce the uniformity (or security) to a smaller observation set. We can regard the selection of $|\boldsymbol{y}| \cdot d$ output shares, excluding cases where all $d+1$ shares of the same output are involved in the selection, as an observation set $\mathcal{O}_d$ and apply Reduction Rule 1 to it. If all such $\mathcal{O}_d$ can be reduced to an empty set, then the output sharing is uniform. If any $\mathcal{O}_d$ cannot be reduced to an empty set, then the XOR Lemma can be used to verify the uniformity of the remaining observation functions in $\mathcal{O}_d$.

The improved algorithm to check uniformity is shown in Algorithm 3. It first generates all possible combinations of $|\boldsymbol{y}| \cdot d$ output shares, excluding cases where all $d+1$ shares of the same output are involved in the combination. This corresponds to the Cartesian product of $\mathcal{Y}_i$ where $1 \leq i \leq |\boldsymbol{y}|$. Then, ReduceF is called to reduce the size of each element $\boldsymbol{f}$ in $\mathcal{Y}_1 \times \mathcal{Y}_2 \times \cdots \times \mathcal{Y}_{|\boldsymbol{y}|}$. If the corresponding observation set $\mathcal{O}$ is reduced to an empty set, then $\boldsymbol{f}$ is uniformly distributed. Otherwise, Prover verifies whether Equation 7 holds to determine if $\boldsymbol{f}$ is uniformly distributed.

Algorithm 3 Algorithm for Uniformity Check

Input: $Sh(\boldsymbol{y})$
Output: True(False): the output sharing is (not) uniform

function CHECKUNIFORMITY($Sh(\boldsymbol{y})$)
  for i from 1 to $|\boldsymbol{y}|$ **do**
    $\mathcal{Y}_i \leftarrow \emptyset$
    for j from 1 to $d+1$ **do**
      $\mathcal{Y}_i \leftarrow \mathcal{Y}_i \cup (Sh(y_i) \setminus \{y_{ij}\})$
    end for
  end for
  for $\boldsymbol{f} \in \mathcal{Y}_1 \times \mathcal{Y}_2 \times \cdots \times \mathcal{Y}_{|\boldsymbol{y}|}$ **do**
    $\mathcal{O} \leftarrow$ the set of nodes corresponding to $\boldsymbol{f}$
    $\mathcal{O}' \leftarrow$ REDUCEF($\mathcal{O}$)
    $\boldsymbol{f}' \leftarrow \bigcup_{n \in \mathcal{O}'} f_n$ ▷ The set of functions to compute output probability
    for $\boldsymbol{\lambda}$ from 1 to $2^{|\boldsymbol{f}'|} - 1$ **do**
      if $\Pr[\boldsymbol{\lambda f}' = 1] \neq \frac{1}{2}$ **then return** False
      end if
    end for
  end for
  return True
end function

The efficiency of this algorithm depends on how much smaller could $\mathcal{O}$ be reduced. In the best case, all $\mathcal{O}$ are reduced to an empty set. In this scenario, the function REDUCEF is called $(d+1)^{|\boldsymbol{y}|}$ times, and no new ROBDD constructions are required. In the worst case, Reduction Rule is not applied at all, resulting in the construction of ROBDDs for $(d+1)^{|\boldsymbol{y}|}(2^{|\boldsymbol{y}|\cdot d} - 1)$ times, which is comparable to $(2^{d+1} - 1)^{|\boldsymbol{y}|} - 1$. However, the worst case is unlikely to happen because many schemes use fresh randomness to maintain the uniformity property.

Nevertheless, we discovered that the uniformity check of SILVER is extremely efficient when the outputs are 2-shared or the implementation did not utilize fresh randomness, as SILVER has optimized the order of constructing ROBDDs to hit the cache more often. In Prover, we utilize this optimization from SILVER and apply Algorithm 3 only when this optimization is not applicable. This situation arises when the number of output shares exceeds 2 and at least one of the outputs have perfect masks.

5 Experiments and Evaluations

In this section, we have collected various open-sourced masked S-box implementations[2] and standard gadgets[3] as benchmarks for evaluations of the proposed methods in this paper and the comparison with state-of-the-art tools. Details about these benchmarks can be found at subsection 5.1.

We present three experiments in this section. The first experiment examined the impact of different variable orderings on SILVER. The second experiment compared the performance of COCOALMA, maskVerif, SILVER, and Prover in verifying S-boxes. IronMask was excluded since it encounters difficulties with these. Specifically, IronMask seems to require that the number of sensitive inputs in the masked implementations be 2 or fewer; exceeding this threshold leads to assertion failures or segmentation faults.

[2]Public available at https://github.com/Lucien98/coco-alma_evaluation/tree/main/examples
[3]From https://github.com/CryptoExperts/IronMask/tree/main/gadgets

The third experiment evaluated the performance of IronMask, maskVerif, SILVER, and Prover in verifying standard gadgets. CocoAlma was excluded since we could not find open-sourced benchmarks (Verilog implementations) for CocoAlma. The results of the three experiments are shown in subsection 5.2, 5.3, and 5.4, respectively. All evaluation benchmarks, scripts, and tool source code are public available under an open-source license[4]. All experiments were performed on an Ubuntu 22.04 TLS virtual machine running on an Intel Core i7-6700 processor clocked at 3.40 GHz. The virtual machine was equipped with 16 GB of memory and 8 logical processor cores. SILVER and Prover utilized all 8 cores, whereas IronMask, CocoAlma and maskVerif only used 1 core.

Finally, we provide a detailed analysis on the insecure S-boxes appeared in our experiments in subsection 5.5.

5.1 Benchmarks

Benchmarks of S-boxes. Table 2 presents details about the masked S-box implementations we collected. It includes information on the reference, abbreviations for the scheme, the number of secret variables $|\boldsymbol{x}|$, the number of shares $|Sh(x)|$, the expected security order d, the number of clock cycles required for computation, the count of fresh masks, the total number of gates $|\mathcal{N}|$, and the number of probing positions under glitch-extended probing model $|\mathcal{P}_g|$. For Prover and SILVER, the verification complexity under standard probing model is $\sum_{i=1}^{d}\binom{|\mathcal{N}|-|\mathcal{P}_g|}{i}$ for a d-th order secure implementation while the complexity under glitch-extended probing model is $\sum_{i=1}^{d}\binom{|\mathcal{P}_g|}{i}$.

In the abbreviation of a scheme, the subscript stands for the expected security order d. The abbreviations for implementations using the techniques from [RBN+15a] and [GMK16] are CMS and DOM. Implementations from [SM21a] (or [SM21b]) that utilize no (or almost no) fresh randomness are denoted as NF_1 (or NF_2), while 1F indicates the use of one bit of fresh randomness. The INF_1 implementation of PRINCE refers specifically to the inverse of the PRINCE S-box. The $\mathrm{4F}_1$ implementation of AES S-box, detailed in [YCW+24], incorporates 4 bit fresh masks and 8-bit guards from neighboring S-boxes. Threshold implementations are marked as TI. The symbol L_d (with only a subscript) stands for the low-latency Keccak implementations in [ZSS+21], while the L_d^c implementations (with subscript and superscript) come from [BDMS22] where d denotes the expected security order and $c+1$ indicates the number of clock cycles to complete the computation. LL denotes low-latency and low-randomness implementations as proposed in [SDM23]. Implementations marked with a star in their abbreviations are paired-versions of masked S-boxes. LL_2 and LL_2^* implementations of SKINNY were excluded from our benchmarks since these implementations failed to encrypt correctly in our simulation.

[4]Evaluation benchmarks and scripts for CocoAlma, maskVerif, IronMask, Prover and SILVER are public available at coco-alma_evaluation, maskVerif_evaluation, IronMask and prover, respectively.

Table 2: Information about masked S-box implementations

Reference	Impl.	$\|\boldsymbol{x}\|$	$Sh(x)$	d	Cycl.	#ref	$\|\mathcal{N}\|$	$\|\mathcal{P}_g\|$
AES								
[SM21a]	$1F_1$	8	2	1	6	1	967	176
[YCW+24]	$4F_1$	8	2	1	6	12	1004	188
[DRB+16]	CMS_1	8	2	1	8	54	938	192
[GMK17]	DOM_1	8	2	1	8	18	884	240
[SM21a]	NF_1	8	2	1	6	0	1188	240
[UHA17]	TI_1	8	2	1	3	64	776	112
Keccak								
[GSM17]	DOM_1	5	2	1	2	5	121	30
[GSM17]	DOM'_1	5	2	1	2	0	111	30
[GSM17]	DOM_2	5	3	2	2	15	249	60
[GSM17]	DOM_3	5	4	3	2	30	462	100
[ZSS+21]	L_1	5	2	1	1	5	85	20
[ZSS+21]	L_2	5	3	2	1	15	180	45
[ZSS+21]	L_3	5	4	3	1	30	310	80
[SM21b]	NF_1	5	2	1	2	0	96	30
[SM21b]	NF_2	5	3	2	2	0	188	60
Midori								
[BDMS22]	L_2^3	4	3	2	4	51	444	91
[BDMS22]	L_2^4	4	3	2	5	24	277	84
[BDMS22]	L_{2*}^3	8	3	2	4	90	864	170
[SDM23]	LL_2	4	3	2	2	104	1189	108
[SDM23]	LL_2^*	8	3	2	2	192	1918	216
[SM21a]	NF_1	4	2	1	2	0	204	36
[SM21b]	NF_2	4	3	2	5	8	328	102
PRESENT								
[BDMS22]	L_2^3	4	3	2	4	53	428	92
[BDMS22]	L_2^5	4	3	2	6	24	299	96
[BDMS22]	L_{2*}^3	8	3	2	4	90	824	168
[SM21a]	NF_1	4	2	1	2	0	178	36
[SM21b]	NF_2	4	3	2	5	8	326	102
[EGMP17]	$TINU_1$	4	3	1	2	0	161	24
[EGMP17]	TIU_1	4	3	1	2	0	177	24
[PMK+11]	TIU'_1	4	3	1	2	0	377	24
PRINCE								
[SM21a]	INF_1	4	2	1	2	0	250	40
[BDMS22]	L_2^4	4	3	2	5	52	497	109
[BDMS22]	$L_{2'}^4$	4	3	2	5	53	498	109
[BDMS22]	L_2^6	4	3	2	7	38	378	120
[SDM23]	LL_2	4	3	2	2	116	1249	120
[SDM23]	LL_2^*	8	3	2	2	216	2140	240
[SM21a]	NF_1	4	2	1	2	0	211	40
[SM21b]	NF_2	4	3	2	8	16	378	138
[MS16]	TI_1	4	3	1	3	0	150	36
SKINNY								
[BJK+20]	CMS_1	8	2	1	5	0	192	96
[BDMS22]	L_2^3	4	3	2	4	36	292	76
[BDMS22]	L_2^4	4	3	2	5	32	272	84
[BDMS22]	L_{2*}^3	8	3	2	4	64	568	144
[SM21b]	NF_2	4	3	2	4	8	202	72
[BJK+16]	TI_1	8	3	1	4	0	240	96

Benchmarks of Standard Gadgets. We considered the following gadgets: the ISW multiplication [ISW03] and $n\log n$ refresh [BCPZ16]. The benchmarks for these gadgets were sourced from IronMask's GitHub repository. This repository includes a script to convert benchmarks for IronMask into formats compatible with SILVER and Prover. Additionally, we wrote a script that converts benchmarks for SILVER and Prover into formats suitable for maskVerif. Unfortunately, we did not find suitable benchmarks for CocoAlma, so it is excluded from this experiment. Furthermore, 3-shared and 5-shared $n\log n$ refresh gadgets are not provided in IronMask's GitHub repository, so these were also excluded.

The detailed information about these gadgets is shown in Table 3. It includes the number of shares $|Sh(x)|$, the expected security order under standard probing model d_s, the expected security ordering under glitch-extended probing model d_g, the count of fresh masks (#ref), the number of gates $|\mathcal{N}|$, and the number of probing positions under glitch-extended probing model ($\mathcal{P}_g$). Note that the ISW multiplication gadgets provided by IronMask's Github repository do not use registers to stop propagation of glitches, meaning none of them is glitch-extended probing secure at any order $d_g \geq 1$.

Table 3: Information about standard gadgets

Gadgets	$\|Sh(x)\|$	d_s	d_g	#ref	$\|\mathcal{N}\|$	$\|\mathcal{P}_g\|$
ISW mult [ISW03]	2	1	0	1	15	2
	3	2	0	3	33	3
	4	3	0	6	58	4
	5	4	0	10	90	5
	6	5	0	15	129	6
refresh $n\log n$ [BCPZ16]	2	1	1	1	7	2
	4	3	3	6	26	4
	6	5	5	12	54	12
	7	6	6	15	64	12

5.2 Evaluations for Different Variable Orderings

In the first experiment, we configured SILVER with different variable orderings to verify the standard probing security of the benchmarks. We opted not to extend this comparison to verifying glitch-extended probing security and uniformity, as their verification would take more than 24 hours on several benchmarks.

Note that in this experiment we only modified the variable orderings of SILVER, i.e., the reduction rules were not employed, to show the influence of different variable orderings. The results are shown in Table 4. There are 6 different columns in Table 4. The first column represents the names of the verified implementations. Columns 2-4 are the verification result and time of SILVER using its original ordering strategy, ordering-1, and ordering-2 strategies under standard probing model, respectively. The symbol ^{d}✓ indicates this implementation is secure at order d. The symbol ^{d}✘ indicates this implementation is not secure at order d. Ordering 1 refers to the optimal variable ordering mentioned in [HSSW10], while ordering 2 declares secret variables in $\boldsymbol{x}$ as ROBDD variables. Columns 5-6 shows the speedup comparing Prover with SILVER, namely the quotient of columns 2/2 and columns 3/4.

In the 45 benchmarks, the optimal ordering outperforms the original ordering in 37 cases, while the second ordering outperforms in 44 cases (marked with bold font in the speedup columns). There are only six cases where the optimal ordering is less efficient than the original one (the speedup is less than 1.0), but it only takes less than one second to verify them. There is only one case where the second ordering does not outperform the original one, and the performance of the second ordering is comparable to the original one in this benchmark.

Due to these findings, we equipped Prover with the ordering-2 strategy in the second experiment.

5.3 Comparison with State-Of-The-Art Tools Over S-boxes

In our second experiment, we compared Prover to three state-of-the-art tools: CocoAlma [HB21], maskVerif [BBC+19] and SILVER [KSM20] over S-box implementations.

Table 5 presents the verification results and time of the four tools applied to these masked implementations. The first column lists the abbreviation of the implementation. Columns 2-5 (6-9) show the verification results and time of CocoAlma, maskVerif, SILVER, and Prover under standard (glitch-extended) probing model. All tools terminate upon encountering the first detected leakage, otherwise they verify all possible observations according to the expected security order d. Columns 10-11 are the uniformity check results and required time of SILVER and Prover. It is noteworthy to mention that we identified a bug in SILVER's uniformity check: when implementing the method from Lemma 1, SILVER failed to include cases where at least one, but not all, of $\boldsymbol{\lambda}^{(i)}$s are zeros. The bug decreases the complexity of uniformity check from $(2^{d+1}-1)^n-1$ to $(2^{d+1}-2)^n$, falsely showing greater efficiency. Moreover, this bug caused SILVER to erroneously report that the output sharing of the NF_2 implementation of PRESENT S-box is uniform (the

Table 4: Comparison between different variables ordering in SILVER over verification of standard probing security

Impl.	Original	Ordering 1	Ordering 2	Speedup ord. 1	Speedup ord. 2
AES					
$1F_1$	[5.3 s]	[2.8 s]	[1.0 s]	**1.89**	**5.3**
$4F_1$	[13 s]	[4.6 s]	[1.1 s]	**2.83**	**11**
CMS_1	[14 s]	[5.8 s]	[1.00 s]	**2.41**	**14**
DOM_1	[12 s]	[4.2 s]	[0.67 s]	**2.86**	**17**
NF_1	[5.7 s]	[2.2 s]	[0.77 s]	**2.59**	**7.4**
TI_1	[10 s]	[4.1 s]	[0.81 s]	**2.44**	**12**
Keccak					
DOM_1	[52 ms]	[66 ms]	[45 ms]	0.79	**1.16**
DOM'_1	[54 ms]	[0.23 s]	[42 ms]	0.23	**1.29**
DOM_2	[4.7 s]	[1.6 s]	[1.3 s]	**2.94**	**3.62**
DOM_3	[1.1 h]	[19 min]	[16 min]	**3.47**	**4.13**
L_1	[46 ms]	[40 ms]	[38 ms]	**1.15**	**1.21**
L_2	[0.66 s]	[0.72 s]	[0.46 s]	0.92	**1.43**
L_3	[3.2 min]	[3.2 min]	[2.4 min]	1.0	**1.33**
NF_1	[51 ms]	[41 ms]	[43 ms]	**1.24**	**1.19**
NF_2	[1.1 s]	[0.91 s]	[0.68 s]	**1.21**	**1.62**
PRESENT					
L_2^3	[21 s]	[14 s]	[15 s]	**1.5**	**1.4**
L_2^5	[7.6 s]	[4.5 s]	[4.5 s]	**1.69**	**1.69**
$L_{2^*}^3$	[2.6 h]	[5.0 min]	[9.0 min]	**31**	**17**
NF_1	[75 ms]	[0.16 s]	[66 ms]	0.47	**1.14**
NF_2	[4.4 s]	[2.6 s]	[1.7 s]	**1.69**	**2.59**
$TINU_1$	[1.3 s]	[55 ms]	[51 ms]	**23**	**25**
TIU_1	[0.21 s]	[0.21 s]	[61 ms]	1.0	**3.44**
TIU'_1	[0.14 s]	[0.12 s]	[91 ms]	**1.17**	**1.54**
SKINNY					
CMS_1	[1.3 s]	[94 ms]	[74 ms]	**13**	**17**
L_2^3	[5.0 s]	[3.4 s]	[3.5 s]	**1.47**	**1.43**
L_2^4	[4.3 s]	[2.8 s]	[3.0 s]	**1.54**	**1.43**
L_{2*}^3	[27 min]	[1.3 min]	[1.9 min]	**20**	**14**
NF_2	[1.5 s]	[0.86 s]	[0.73 s]	**1.74**	**2.05**
TI_1	[1.5 min]	[0.44 s]	[1.5 s]	**204**	**60**
PRINCE					
INF_1	[58 ms]	[1.0 s]	[55 ms]	0.06	**1.05**
L_2^4	[50 s]	[49 s]	[44 s]	**1.02**	**1.14**
$L_{2'}^4$	[8.1 min]	[6.3 min]	[6.0 min]	**1.29**	**1.35**
L_2^6	[7.2 min]	[6.9 min]	[6.3 min]	**1.04**	**1.14**
LL_2	[2.5 min]	[2.1 min]	[2.5 min]	**1.19**	1.0
LL_2^*	[8.5 h]	[30 min]	[56 min]	**17**	**9.11**
NF_1	[55 ms]	[0.83 s]	[54 ms]	0.07	**1.02**
NF_2	[41 s]	[32 s]	[26 s]	**1.28**	**1.58**
TI_1	[66 ms]	[52 ms]	[49 ms]	**1.27**	**1.35**
Midori					
L_2^3	[16 s]	[9.3 s]	[9.7 s]	**1.72**	**1.65**
L_2^4	[6.1 s]	[3.4 s]	[3.6 s]	**1.79**	**1.69**
$L_{2^*}^4$	[2.5 h]	[4.2 min]	[5.5 min]	**35**	**27**
LL_2	[2.1 min]	[1.8 min]	[2.0 min]	**1.17**	**1.05**
LL_2^*	[7.2 h]	[23 min]	[39 min]	**18**	**11**
NF_1	[58 ms]	[55 ms]	[55 ms]	**1.05**	**1.05**
NF_2	[4.2 s]	[2.7 s]	[1.6 s]	**1.56**	**2.62**

authors of this implementation also claimed uniformity [SM21b]), whereas it is not. We have corrected this bug in SILVER and the results in Table 5 reflect the corrected version.

The symbols ✓ and ✘ have the same meanings as described in subsection 5.2. CocoAlma and maskVerif suffer from false positives, where they may incorrectly categorize a secure implementation as an insecure one. Such cases are marked with the symbol ✘. In several benchmarks, SILVER exceeds the 24-hour time limit and CocoAlma encounters memory issues. We use the symbol $?_d$ to denote such cases, indicating that the tool does not complete the verification, and we are unsure whether it could verify that this implementation is d-th order secure with more allotted time and memory. The symbols ✓, ✘, and ? indicate the implementation is uniform, not uniform, or that the tool could not verify uniformity within the time limit.

5.3.1 Comparison over Four Tools

CocoAlma and maskVerif are more efficient formal verification tools than SILVER, but both have false positives. We now briefly introduce these two tools.

Table 5: Verification results and required time by COCO-ALMA, maskVerif, SILVER, and Prover

	Standard Probing Security				Glitch-extended Probing Security				Uniformity	
	coco-alma [HB21]	maskVerif [BBC+19]	SILVER [KSM20]	Prover *this work*	coco-alma [HB21]	maskVerif [BBC+19]	SILVER [KSM20]	Prover *this work*	SILVER [KSM20]	Prover *this work*
AES										
$1F_1$	✘ 0.57 s	✘ 0.88 s	✓ 6.1 s	✓ **1.1 s**	✘ 1.5 s	✘ 26 ms	✓ 26 s	✓ **3.9 s**	✘ 35 s	✘ **4.6 s**
$4F_1$	✘ 1.1 s	✘ 0.21 s	✓ 14 s	✓ **1.5 s**	✘ 1.9 s	✘ 56 s	✓ 1.3 h	✓ **[9.1 min]**	✓ 5.9 min	✓ **9.0 s**
CMS_1	✓ 5.3 s	✓ 1.6 s	✓ 16 s	✓ **0.91 s**	✓ 7.5 s	✓ **0.41 s**	✓ 4.7 h	✓ 1.7 s	✓ 11 min	✓ **22 s**
DOM_1	✓ 1.8 s	✓ **0.54 s**	✓ 12 s	✓ 0.68 s	✓ 12 s	✓ **63 ms**	✓ 1.9 h	✓ 6.0 s	✓ 10 min	✓ **27 s**
NF_1	✘ 0.26 s	✘ 0.26 s	✓ 4.5 s	✓ **1.0 s**	✘ 0.54 s	✘ 22 ms	✓ 18 s	✓ **2.5 s**	✘ 42 s	✘ **15 s**
TI_1	✓ 0.54 s	✓ **0.13 s**	✓ 12 s	✓ 0.76 s	✓ 0.56 s	✓ **35 ms**	$?_1$ > 24 h	✓ 1.6 s	? > 24 h	✓ **14 min**
Keccak										
DOM_1	✓ 4.7 ms	✓ **2.0 ms**	✓ 55 ms	✓ 72 ms	✓ 10 ms	✓ **2.0 ms**	✓ 41 ms	✓ 0.16 s	✓ **0.13 s**	✓ 0.59 s
DOM'_1	✓ 4.2 ms	✓ **3.0 ms**	✓ 56 ms	✓ 80 ms	✓ 8.8 ms	✓ **2.0 ms**	✓ 34 ms	✓ 0.11 s	✘ **62 ms**	✘ 0.85 s
DOM_2	✓ 3.0 s	✓ **22 ms**	✓ 3.7 s	✓ 0.61 s	✓ 0.62 s	✓ **27 ms**	✓ 7.4 s	✓ 0.44 s	✓ 53 s	✓ **1.7 s**
DOM_3	$?_3$ OoM	✓ **1.0 s**	✓ 1.1 h	✓ 14 min	✓ 32 min	✓ **0.65 s**	✓ 7.1 h	✓ 1.6 min	✓ 2.7 h	✓ **3.3 s**
L_1	✓ 3.7 ms	✓ **2.0 ms**	✓ 42 ms	✓ 54 ms	✓ 4.1 ms	✓ **1.0 ms**	✓ 29 ms	✓ 0.15 s	✘ **0.39 s**	✘ 1.6 s
L_2	✓ 0.70 s	✓ **5.0 ms**	✓ 0.67 s	✓ 0.26 s	✓ 0.15 s	✓ **4.0 ms**	✓ 1.6 s	✓ 0.32 s	✘ **0.44 s**	✘ 0.49 s
L_3	$?_3$ OoM	✓ **88 ms**	✓ [3.3 min]	✓ [1.6 min]	✓ [2.5 min]	✓ **36 ms**	✓ 16 min	✓ 14 s	✘ **0.38 s**	✘ 0.79 s
NF_1	✓ 5.5 ms	✓ **2.0 ms**	✓ 49 ms	✓ 64 ms	✓ 8.5 ms	✓ **2.0 ms**	✓ 32 ms	✓ 94 ms	✓ **0.32 s**	✓ 1.1 s
NF_2	✓ 19 ms	✓ **0.12 s**	✓ 1.3 s	✓ 0.30 s	✓ 20 ms	✓ 3.0 ms	✓ 2.2 s	✓ **0.36 s**	✓ 3.6 s	✓ **2.5 s**
Midori										
L_2^3	✓ [3.7 min]	✓ **0.29 s**	✓ 16 s	✓ 4.0 s	✓ 12 s	✓ **0.12 s**	✓ 5.4 h	✓ 11 s	✓ 37 s	✓ **5.6 s**
L_2^4	✓ 36 s	✓ **0.11 s**	✓ 5.8 s	✓ 0.99 s	✓ 5.9 s	✓ **96 ms**	✓ 39 s	✓ 1.4 s	✓ 22 s	✓ **0.62 s**
$L_{2^*}^3$	✓ 36 min	✓ **1.9 s**	✓ 2.8 h	✓ 25 s	✓ [3.5 min]	✓ **1.0 s**	$?_2$ > 24 h	✓ 1.4 min	? > 24 h	✓ **21 s**
LL_2	✓ [5.1 min]	✓ **3.3 s**	✓ [2.1 min]	✓ [1.2 min]	✓ 7.2 s	✓ **0.37 s**	$?_2$ > 24 h	✓ 11 s	✓ 5.4 s	✓ **0.93 s**
LL_2^*	$?_2$ OoM	✓ **18 s**	✓ 7.7 h	✓ [4.5 min]	✓ 53 s	✓ **3.2 s**	$?_2$ > 24 h	✓ 47 s	? > 24 h	✓ **24 s**
NF_1	✘ 23 ms	✘ 8.0 ms	✓ 0.12 s	✓ **57 ms**	✘ 30 ms	✘ 8.0 ms	✓ 0.13 s	✓ **0.10 s**	✓ 0.54 s	✓ **0.47 s**
NF_2	✘ 0.11 s	✘ 0.90 s	✓ 4.4 s	✓ **1.8 s**	✘ 95 ms	✘ 11 ms	✓ 11 s	✓ **2.4 s**	✓ 8.5 s	✓ **0.30 s**
PRESENT										
L_2^3	✓ [2.8 min]	✓ **0.50 s**	✓ 21 s	✓ 3.4 s	✓ 8.5 s	✓ **0.18 s**	$?_2$ > 24 h	✓ 23 s	✓ 41 s	✓ **5.5 s**
L_2^5	✓ [1.5 min]	✓ **0.34 s**	✓ 7.5 s	✓ 1.5 s	✓ 12 s	✓ **0.18 s**	✓ 47 s	✓ 1.9 s	✓ 26 s	✓ **1.4 s**
$L_{2^*}^3$	✓ [24 min]	✓ **4.8 s**	✓ 2.7 h	✓ 20 s	✓ [2.4 min]	✓ **1.7 s**	$?_2$ > 24 h	✓ [6.1 min]	? > 24 h	✓ **47 s**
NF_1	✘ 18 ms	✘ 6.0 ms	✓ **59 ms**	✓ 0.19 s	✘ 18 ms	✘ 8.0 ms	✓ **53 ms**	✓ 0.27 s	✓ 1.7 s	✓ **87 ms**
NF_2	✘ 89 ms	✘ 1.7 s	✓ 4.4 s	✓ **2.3 s**	✘ 88 ms	✘ 6.0 ms	✓ 13 s	✓ **3.9 s**	✘ 4.2 s	✘ **56 ms**
$TINU_1$	✘ 16 ms	✘ 65 ms	✘ **64 ms**	✘ 0.63 s	✘ **23 ms**	✘ 5.0 ms	✘ 72 ms	✘ 0.75 s	✘ **0.18 s**	✘ 0.57 s
TIU_1	✘ 24 ms	✘ 0.26 s	✓ 77 ms	✓ **57 ms**	✘ 24 ms	✘ 7.0 ms	✓ **97 ms**	✓ 0.12 s	✓ 0.28 s	✓ **14 ms**
TIU'_1	✘ 19 ms	✘ 67 ms	✓ **0.14 s**	✓ 0.15 s	✘ 40 ms	✘ 18 ms	✓ **0.12 s**	✓ 0.24 s	✓ 0.26 s	✓ **0.11 s**
PRINCE										
INF_1	✘ 31 ms	✘ 11 ms	✓ **67 ms**	✓ 84 ms	✘ 40 ms	✘ 7.0 ms	✓ **58 ms**	✓ 0.14 s	✘ **0.37 s**	✘ 0.53 s
L_2^4	✘ [13 min]	✘ **1.7 s**	✘ 50 s	✘ 6.3 s	✘ 0.69 s	✘ 0.39 s	✘ [39 min]	✘ **[2.0 min]**	✓ 2.6 min	✓ **[2.3 min]**
$L_{2'}^4$	✘ 14 min	✓ **1.7 s**	✓ [7.1 min]	✓ 18 s	✘ 0.72 s	✘ 0.41 s	$?_2$ > 24 h	✓ **13 min**	✓ 3.8 min	✓ **[2.3 min]**
L_2^8	✓ [6.8 min]	✓ **1.00 s**	✓ [6.3 min]	✓ 8.2 s	✘ 6.2 s	✘ 47 min	✓ 4.0 h	✓ **18 s**	✓ **[2.8 min]**	✓ 28 min
LL_2	$?_2$ OoM	✓ **3.2 s**	✓ [2.6 min]	✓ [1.3 min]	✓ 50 s	✓ **0.43 s**	$?_2$ > 24 h	✓ 1.6 min	✓ 18 s	✓ **0.33 s**
LL_2^*	$?_2$ OoM	✓ **22 s**	✓ 9.1 h	✓ [6.2 min]	✓ [1.4 min]	✓ **4.2 s**	$?_2$ > 24 h	✓ 26 min	? > 24 h	✓ **34 s**
NF_1	✘ 24 ms	✘ 13 ms	✘ **64 ms**	✘ 68 ms	✘ 40 ms	✘ 7.0 ms	✘ **55 ms**	✘ 0.15 s	✘ **0.20 s**	✘ 1.7 s
NF_2	✘ 0.28 s	✘ 17 s	✓ 45 s	✓ **5.0 s**	✘ 0.21 s	✘ 6.0 ms	✓ [2.2 min]	✓ **8.4 s**	✓ 22 s	✓ **0.98 s**
TI_1	✘ 24 ms	✘ 27 ms	✓ 70 ms	✓ **51 ms**	✘ 53 ms	✘ 16 ms	✓ 0.14 s	✓ **0.12 s**	✓ 0.78 s	✓ **0.52 s**
SKINNY										
CMS_1	✘ 25 ms	✘ 37 ms	✓ 0.36 s	✓ **65 ms**	✘ 24 ms	✘ 10 ms	✓ 0.89 s	✓ **0.13 s**	✓ 1.7 s	✓ **0.74 s**
L_2^3	✓ 34 s	✓ **78 ms**	✓ 4.9 s	✓ 0.98 s	✓ 3.4 s	✓ **70 ms**	✓ [18 min]	✓ 2.1 s	✓ 20 s	✓ **5.3 s**
L_2^4	✓ 27 s	✓ **64 ms**	✓ 4.2 s	✓ 0.73 s	✓ 3.4 s	✓ **49 ms**	✓ 27 s	✓ 0.81 s	✓ 21 s	✓ **0.36 s**
$L_{2^*}^3$	✓ [3.9 min]	✓ **0.49 s**	✓ [30 min]	✓ 6.0 s	✓ 30 s	✓ **0.49 s**	$?_2$ > 24 h	✓ 9.8 s	? > 24 h	✓ **16 s**
NF_2	✘ 30 ms	✘ 56 ms	✓ 1.7 s	✓ **0.38 s**	✘ 51 ms	✘ 4.0 ms	✓ 3.9 s	✓ **0.57 s**	✓ 10 s	✓ **0.22 s**
TI_1	✘ 41 ms	✘ [3.5 min]	✓ [1.4 min]	✓ **1.4 s**	✘ 50 ms	✘ 1.1 h	✓ [3.5 min]	✓ **3.4 s**	✓ 18 min	✓ 18 min

Introduction to CocoAlma. CocoAlma represents an upgraded version of REBECCA [BGI+18]with enhanced usability and performance, supporting stateful hardware verification. The verification process in CocoAlma unfolds through three primary steps. Initially, it parses the provided hardware design into a gate-level Verilog netlist using Yosys [WGK13]. Subsequently, Verilator [Sny04] is employed to simulate this netlist alongside a user-provided testbench. We used this step to determine the inclusion of each open-source masked implementation collected for our benchmarks. For example, implementations that fail to produce correct outputs, such as the LL implementations of SKINNY S-boxes from [SDM23], were excluded from our evaluation. The third step is to verify the side-channel security of a masked implementation in a specified security model.

CocoAlma supports three models: standard probing model (or *software* probing model in their terminology), *time-constrained* probing model, and *stateful hardware* probing model. In time-constrained probing model, CocoAlma accounts for leakage not only from glitches but also from register transitions. Register transitions often happen when the implementation processes two consecutive inputs during two consecutive clock cycles.

However, in our experiments, the inputs are fed to the circuit only in the first cycle when using CocoAlma. We believe that the transitional leakage is unlikely to happen in this setup thus the comparison to CocoAlma is relatively fair. The third model can handle stateful circuits, a capability not supported by maskVerif, SILVER, and Prover. Therefore, we used CocoAlma to verify the masked implementations only under the *software* probing model and *time-constrained* probing model. The results under these two models are shown in the standard probing security column and glitch-extended probing security column of Table 5, respectively.

Another distinction between CocoAlma and other tools is its use of an execution trace that simulates the hardware running process. This approach is more realistic but requires the user to identify the number of clock cycles to verify. We configured this parameter based on the *Cycl.* column in Table 2.

Introduction to maskVerif. maskVerif is an efficient verification tool that utilizes features of programming language. It employs three rules – INDEP, OPT, and CONV – to verify the security of a masked hardware implementation. Rule INDEP asserts that if no secret variable appears in a expression e (observable function f in our terminology), then e is deemed secure. Rule OPT operates similarly to our Reduction Rule 1. It stipulates that if $n \in \mathcal{O}_d$ and $\mathrm{perf}(n) \cap \mathrm{supp}(\mathcal{O}_d \setminus \{n\}) = \emptyset$, then substitute the expression of f_n with $r \in \mathrm{perf}(n)$ in $\mathcal{O}_d$, obtaining $\mathcal{O}'_d$ and continue verification with $\mathcal{O}'_d$. Rule CONV employs algebraic normalization to simplify expressions. Take the threshold implementation for $\mathcal{Q}_{12}$ in Appendix A.2 of [RBN+15b] for example. Consider a single observation $\bar{z}_1 = z_1 + z_2 = (a_1b_1 + a_1c_1 + c_1) + (a_1c_2 + a_1b_2)$ where a, b, and c are the secret variable and $a_1, a_2, b_1, b_2, c_1, c_2$ are their corresponding shares. Neither rule OPT nor Reduction Rule 1 are directly applicable to verify the security of $\bar{z}_1$. However, maskVerif can verify it using rule CONV. In maskVerif's internal representation, a_1, b_1, c_1 are treated as uniform random variables, and a_2, b_2, c_2 are represented as $a+a_1, b+b_1, c+c_1$. Normalizing $\bar{z}_1$ results in $\bar{z}_1 = a_1b_1 + a_1c_1 + c_1 + a_1(c + c_1) + a_1(b + b_1) = c_1 + a_1c + a_1b$. Now, c_1 acts as a perfect mask for $\bar{z}_1$, allowing application of rule OPT or Reduction Rule 1 to conclude that z_1 is secure. The application of rule CONV is likely the reson why maskVerif has fewer false positive cases under the standard probing model compared to CocoAlma. In fact, maskVerif could identify that the $\mathcal{Q}_{12}$ is standard probing secure due to rule CONV while CocoAlma can not. Nevertheless, as noted by the authors of maskVerif, rule CONV negatively impacts performance [BBD+15], which results in slower verification times for certain false positive cases in our experiments, such as the TI_1 implementation of the SKINNY S-box.

Identification of False Positives. Out of 45 benchmarks, there are 43 benchmarks that are secure under both models. For the secure implementations, any leakage reported by CocoAlma and maskVerif is considered as a false positive. CocoAlma and maskVerif exhibit a significant issue with false positives when verifying Threshold Implementations or implementations from [SM21a] and [SM21b]. These implementations rely on Boolean function properties rather than fresh randomness to achieve standard or glitch-extended probing security, while CocoAlma and maskVerif appear to excel in verifying implementations that achieve security through fresh randomness.

Regarding the two insecure implementations, detailed analyses are available in the repository of Prover[5] to determine whether the leakages reported by CocoAlma and maskVerif are real or false positives. Ultimately, we found that CocoAlma correctly identified the real leakage of TINU_1 implementation of PRESENT S-box under glitch-extended probing model and maskVerif correctly identified the real leakage of L_2^4 implementation of PRINCE S-box under standard probing model. However, other reported leakages on these two implementations by CocoAlma and maskVerif were false positives.

Comparison on Usability and Accuracy. CocoAlma encounters memory issues

[5] https://github.com/Lucien98/prover/tree/uniformity/experiment/tches2025_1

in 5 benchmarks, primarily due to their higher order (such as the two third-order secure implementations of the Keccak S-box) or larger circuit sizes (e.g., the paired version of masked S-boxes).

In contrast to CocoAlma and maskVerif, SILVER avoids false positive cases by utilizing Binary Decision Diagrams for symbolic and exhaustive analysis of probability distributions and statistical independence of joint distributions. However, SILVER's performance is notably slower compared to other tools, especially with larger circuit sizes. For instance, SILVER requires more than one hour to verify 5 benchmarks out of 45 under the standard probing model, whereas other tools complete in at most 36 minutes. Its performance under glitch-extended probing models is even more constrained, with timeouts occurring in 10 benchmarks. It also runs out of time on 6 benchmarks in uniformity check.

Table 6 provides an overview of the number of successfully verified benchmarks and failures due to various reasons for each tool. CocoAlma and maskVerif handle a total of 90 verification instances, whereas SILVER and Prover manage 135, as SILVER and Prover verify an additional security notion compared to CocoAlma and maskVerif in our experiments. CocoAlma, maskVerif, and SILVER experience failures in 50%, 42%, and 12% of verification instances, respectively, while Prover succeeds in all instances. Maximum verification time is also compared across tools, revealing that Prover has the shortest maximum verification time.

Table 6: Performance comparison between COCO-ALMA, maskVerif, SILVER, and Prover

Tool	#False Pos.		#OoM	#Timout			#Succ.	Max. Time
	std.	rob.		std.	rob.	unif.		
CocoAlma	20	20	5	0	0	N/A	45/90	36 min
maskVerif	17	21	0	0	0	N/A	52/90	1.1 h
SILVER	0	0	0	0	10	6	119/135	$>$ 24 h
Prover	0	0	0	0	0	0	135/135	28 min

Comparison over Non-false-positives. Now we compare the four tools on the benchmarks that no tool exhibits false positives. We begin the comparison with the twenty-five benchmarks under standard probing model. maskVerif performs best on 24 benchmarks, with Prover achieving the best result on one benchmark. Overall, Prover generally outperforms CocoAlma and SILVER. However, Prover does not surpass CocoAlma and SILVER in very small circuits, such as the four 1st-order implementations of the Keccak S-box. Given the small circuit size, the methods employed by other tools are efficient enough. However, Prover not only dedicates time to computing ROBDD representations for observable functions but also computes auxiliary data structures. Therefore, it is understandable that Prover requires more time. Prover also does not outperform CocoAlma in the TI_1 implementation of AES S-box, but Prover completes its verification in less than 1 second.

This pattern persists in the verification of 24 true negative cases under glitch-extended probing model. However, Prover does not outperform maskVerif in any benchmark this time. Apart from the previously mentioned 5 benchmarks, CocoAlma outperforms Prover on 6 larger implementations (Keccak L_2, Midori LL_2, PRESENT L_2^3 and $L_{2^*}^3$, PRINCE LL_2 and LL_2^*). Conversely, Prover outperforms CocoAlma in the rest 13 benchmarks.

5.3.2 Comparison to SILVER

Comparison to SILVER over False Positives of CocoAlma and maskVerif. Under standard probing model, maskVerif fails to verify the security of 17 benchmarks while CocoAlma fails on 20. There are three cases that CocoAlma fails to verify while maskVerif succeeds: NF_2 implementation of Keccak S-box and L_2^4, and $L_{2'}^4$ implementation

of PRINCE S-box. maskVerif performs the best in these three benchmarks among the three tools that could identify their security. Out of the 17 false positives of maskVerif, Prover outperforms SILVER on 12 benchmarks. Under glitch-extended probing model, there are 20 and 21 false positives for CocoAlma and maskVerif respectively. In the case where maskVerif fails to verify while CocoAlma succeeds, i,e., the TINU_1 implementation of PRESENT S-box, CocoAlma performs better than SILVER and Prover. Prover outperforms SILVER on 15 of remaining 20 benchmarks. Under both models, all the benchmarks where SILVER outperforms Prover are verified within 0.5 second by Prover. Note that SILVER runs out of time while attempting to verify the $\text{L}_2^{4'}$ implementation of PRINCE S-box, where CocoAlma and maskVerif encounter false positive issues. In contrast, Prover successfully verifies it within 13 minutes.

Comparison to SILVER on Uniformity Check. Algorithm 3 is not employed when the outputs of these implementations lack perfect masks or are 2-shared, totaling 22 benchmarks. For these benchmarks, the verification is conducted using the original method from SILVER but with a different variable ordering, specifically variable ordering 2 as outlined in this paper. For the remaining benchmarks, Prover utilizes Algorithm 3 to verify their uniformity. There are only 11 cases where Prover does not outperform SILVER. Nine of them can be verified within 2 seconds, which we consider acceptable. In the remaining two case, one is comparable to SILVER, while the other can be verified within half an hour. Notably, half an hour represents the longest verification time of Prover in all our experiments. On the contrary, SILVER runs out of time in 6 benchmarks.

Among the 10 benchmarks that fail to pass the uniformity check, four implementations (AES S-boxes and PRINCE S-boxes) are from [SM21a], which the authors of [SM21a] also claimed they are not uniform. Regarding the NF_2 Implementations of PRESENT, the authors of [SM21b] claimed that it is uniform and the original SILVER falsely recognized it as uniform due to a bug. However, Prover and the corrected SILVER report it as not uniform. The authors of [SM21b] claimed the first-order DOM implementation without fresh randomness (DOM_1' implementation of Keccak) in [GSM17] is not uniform and they provided a uniform solution (NF_1) by using their searching techniques. All these are also confirmed by Prover. As for the L implementations of Keccak S-box, the outputs are not compressed into $d+1$ shares in the part of implementations we synthesized, i.e., each output has $(d+1)^2$ output shares. With $(d+1)^2$ output shares, they fail to pass the uniformity check.

Why Does SILVER Run Out of Time on These Benchmarks? SILVER fails to provide results on glitch-extended probing security and uniformity in several benchmarks due to the large size of observable sets. For instance, the TI_1 implementation of AES S-box has 8 observation sets of size 23 and 32 sets of size 21. Each set of size 23 requires approximately 44 hours to be verified, and each set of size 21 requires about 9 hours. In the LL_2^* implementations, the largest sets are of size 18. There are 153 and 276 such sets in the LL_2^* S-box implementations of Midori and PRINCE, respectively. While we did not further inspect how much time is needed to verify such sets, they are the primary factor contributing to SILVER's slow performance. After applying Reduction Rule 1 of Prover, the largest sets (and the number of largest sets) in these implementations are of size 9 and 16 (with numbers 126 and 10). Since the size of the largest observable sets that need to be verified by ROBDDs is significantly smaller, Prover requires significantly less time to verify these implementations. The same analysis could be applied to the other benchmarks SILVER runs out of time.

The TI_1 implementation of AES S-box does not compress the output shares, and the outputs are 4-shared. To check the uniformity of its output sharing, SILVER needs to construct $(2^4-1)^8-1 \approx 2^{31.3}$ ROBDDs and verify if they are all balanced. With such complexity, it is not surprising that SILVER runs out of time. As for the uniformity check of LL_2^* implementations, SILVER needs to construct $(2^3-1)^8-1 \approx 2^{22.5}$ ROBDDs, which

also leads to running out of time.

Table 7: Number of applications of reduction rules

Scheme	std				rob				
	#Total	#Red.	#Robdd	%Red.	#Total	#Sub.	#Red.	#Robdd	%Red.
AES									
$1F_1$	791	121	670	15%	176	0	80	96	45%
$4F_1$	816	158	658	19%	188	8	84	96	45%
CMS_1	746	458	288	61%	192	0	176	16	92%
DOM_1	644	356	288	55%	240	8	164	68	68%
NF_1	948	77	871	8%	240	4	96	140	40%
TI_1	664	375	289	56%	112	0	112	0	100%
Keccak									
DOM_1	91	71	20	78%	30	0	30	0	100%
DOM_1'	81	51	30	63%	30	0	20	10	67%
DOM_2	17955	10265	7690	57%	1830	69	1650	111	90%
DOM_3	7906623	3626245	4280378	46%	166750	7384	137460	21906	82%
L_1	65	45	20	69%	20	0	20	0	100%
L_2	9180	4010	5170	44%	1035	0	1035	0	100%
L_3	2028025	540440	1487585	27%	85400	0	85400	0	100%
NF_1	66	46	20	70%	30	0	22	8	73%
NF_2	8256	3294	4962	40%	1830	161	1035	634	57%
Midori									
L_2^3	62481	39478	23003	63%	4186	244	3018	924	72%
L_2^4	18721	10371	8350	55%	3570	487	2413	670	68%
$L_{2^*}^4$	241165	151239	89926	63%	14535	663	10639	3233	73%
LL_2	584821	221554	363267	38%	5886	0	5790	96	98%
LL_2^*	1449253	605043	844210	42%	23436	0	23244	192	99%
NF_1	168	27	141	16%	36	0	28	8	78%
NF_2	25651	6174	19477	24%	5253	537	3203	1513	61%
PRESENT									
L_2^3	56616	38140	18476	67%	4278	239	3215	824	75%
L_2^5	20706	11702	9004	57%	4656	333	3408	915	73%
$L_{2^*}^3$	215496	143731	71765	67%	14196	767	10606	2823	75%
NF_1	142	37	105	26%	36	0	30	6	83%
NF_2	25200	8789	16411	35%	5253	484	3200	1569	61%
$TINU_1$	126	60	66	48%	7	0	6	1	86%
TIU_1	263	128	135	49%	26	0	13	13	50%
TIU_1'	570	115	455	20%	25	0	12	13	48%
PRINCE									
INF_1	210	37	173	18%	40	0	32	8	80%
L_2^4	47860	31038	16822	65%	265	38	165	62	62%
$L_{2'}^4$	75855	45548	30307	60%	5995	800	4096	1099	68%
L_2^6	33411	18628	14783	56%	7260	406	5126	1728	71%
LL_2	637885	265322	372563	42%	7260	208	6501	551	90%
LL_2^*	1805950	781301	1024649	43%	1844	375	675	794	37%
NF_1	171	42	129	25%	40	0	32	8	80%
NF_2	28920	11673	17247	40%	9591	433	6654	2504	69%
TI_1	149	48	101	32%	37	15	15	7	41%
SKINNY									
CMS_1	96	51	45	53%	96	12	66	18	69%
L_2^3	23436	16240	7196	69%	2926	22	2673	231	91%
L_2^4	17766	10679	7087	60%	3570	46	2793	731	78%
$L_{2^*}^3$	90100	62024	28076	69%	10440	45	9787	608	94%
NF_2	8515	4051	4464	48%	2628	135	1761	732	67%
TI_1	274	149	125	54%	102	3	93	6	91%

The Application of Reduction Rules and Subset Strategy. For all the benchmarks, we have counted the number of sets verified by Reduction Rules, subset strategy, and statistical independence check via ROBDDs, as shown in Table 7. The first column contains the name of the implementations. Columns 2-5 (and 6-10) show the total sets verified by Prover (starting with $d = 1$), the number of secure sets verified using only reduction rules (as in lines 11 to 15 in Algorithm 2), the number of sets verified by ROBDDs (in the for loop at line 28 in Algorithm 2), and the proportion of sets that are verified solely relying on reduction rules under standard (glitch-extended) probing model. Since the observation sets under glitch-extended probing model have an arbitrary size, the subset strategy is applicable in this situation, and the number of sets verified by this strategy is shown in column 7. From the table, one can see that in 32 benchmarks (more than $\frac{2}{3}$ of benchmarks), more than 40% of sets are verified solely by reduction rules under the standard probing model. This holds true for all benchmarks under the glitch-extended probing model as well, demonstrating the efficacy of reduction rules.

5.4 Comparison with State-Of-The-Art Tools over Standard Gadgets

In our third experiment, we compared Prover with three state-of-the-art tools on standard gadgets, namely IronMask [BMRT22], maskVerif [BBC+19] and SILVER [KSM20].

Introduction to IronMask. IronMask is an automatic verification tool designed to check probing and random probing security properties using complete and efficient procedures. It can verify not only probing security but also random probing composability and expandability. A notable feature of IronMask is its capability to verify standard gadgets at very high orders, even greater than 10. However, despite its claims of completeness in the robust probing model, our experiments reveal that these claims are not entirely accurate.

Verification Results. Table 8 presents the verification results and time of the four tools applied to standard gadgets. The first column lists the names of the standard gadgets, and the second column shows the number of shares for inputs and outputs in these gadgets. Columns 3-6 and 7-10 display the verification results and time for IronMask, maskVerif, SILVER, and Prover under the standard and glitch-extended probing models, respectively. The symbol $\overset{1}{\checkmark}$, which is not used in previous sections, indicates the tool incorrectly reports an insecure implementation as secure at order d. Other symbols in Table 8 have meanings consistent with those described earlier. The abbreviation S. F. in column 3 indicates that IronMask encounters a segmentation fault during verification. The abbreviation OoM in column 6 indicates Prover runs out of memory during verification.

Table 8: Verification results and required time by IronMask, maskVerif, SILVER, and Prover over standard gadgets

Gadgets	$\lvert Sh(x)\rvert$	Standard Probing Security				Glitch-extended Probing Security			
		IronMask [BMRT22]	maskVerif [BBC+19]	SILVER [KSM20]	Prover *this work*	IronMask [BMRT22]	maskVerif [BBC+19]	SILVER [KSM20]	Prover *this work*
ISW mult [ISW03]	2	$\overset{?_1}{\checkmark}$ [S. F.]	$\overset{1}{\checkmark}$ [2.0 ms]	$\overset{1}{\checkmark}$ [12 ms]	$\overset{1}{\checkmark}$ [0.24 ms]	$\overset{1}{\times}$ [< 1 s]	$\overset{1}{\times}$ [1.0 ms]	$\overset{1}{\times}$ [17 ms]	$\overset{1}{\times}$ [12 ms]
	3	$\overset{2}{\checkmark}$ [< 1 s]	$\overset{2}{\checkmark}$ [3.0 ms]	$\overset{2}{\checkmark}$ [64 ms]	$\overset{2}{\checkmark}$ [6.3 ms]	$\overset{1}{\checkmark}$ [< 1 s]	$\overset{1}{\times}$ [1.0 ms]	$\overset{1}{\times}$ [33 ms]	$\overset{1}{\times}$ [31 ms]
	4	$\overset{3}{\checkmark}$ [< 1 s]	$\overset{3}{\checkmark}$ [25 ms]	$\overset{3}{\checkmark}$ [0.47 s]	$\overset{3}{\checkmark}$ [3.2 s]	$\overset{1}{\times}$ [< 1 s]	$\overset{1}{\times}$ [1.0 ms]	$\overset{1}{\times}$ [55 ms]	$\overset{1}{\times}$ [59 ms]
	5	$\overset{4}{\checkmark}$ [< 1 s]	$\overset{4}{\checkmark}$ [0.33 s]	$\overset{4}{\checkmark}$ [1.2 min]	$\overset{4}{\checkmark}$ [5.9 min]	$\overset{1}{\checkmark}$ [< 1 s]	$\overset{1}{\times}$ [2.0 ms]	$\overset{1}{\times}$ [93 ms]	$\overset{1}{\times}$ [0.10 s]
	6	$\overset{5}{\checkmark}$ [< 1 s]	$\overset{5}{\checkmark}$ [6.2 s]	$\overset{5}{\checkmark}$ [3.6 h]	$\overset{?_5}{\checkmark}$ [OoM]	$\overset{1}{\times}$ [< 1 s]	$\overset{1}{\times}$ [4.0 ms]	$\overset{1}{\times}$ [0.42 s]	$\overset{1}{\times}$ [54 ms]
refresh $n\log n$ [BCPZ16]	2	$\overset{1}{\checkmark}$ [< 1 s]	$\overset{1}{\checkmark}$ [1.0 ms]	$\overset{1}{\checkmark}$ [3.9 ms]	$\overset{1}{\checkmark}$ [0.20 ms]	$\overset{1}{\checkmark}$ [< 1 s]	$\overset{1}{\checkmark}$ [1.0 ms]	$\overset{1}{\checkmark}$ [5.1 ms]	$\overset{1}{\checkmark}$ [1.9 ms]
	4	$\overset{3}{\checkmark}$ [< 1 s]	$\overset{3}{\checkmark}$ [2.0 ms]	$\overset{3}{\checkmark}$ [81 ms]	$\overset{3}{\checkmark}$ [40 ms]	$\overset{3}{\checkmark}$ [< 1 s]	$\overset{3}{\checkmark}$ [1.0 ms]	$\overset{3}{\checkmark}$ [62 ms]	$\overset{3}{\checkmark}$ [23 ms]
	6	$\overset{5}{\checkmark}$ [< 1 s]	$\overset{5}{\checkmark}$ [3.0 ms]	$\overset{5}{\checkmark}$ [21 s]	$\overset{5}{\checkmark}$ [1.5 min]	$\overset{5}{\checkmark}$ [< 1 s]	$\overset{5}{\checkmark}$ [2.0 ms]	$\overset{5}{\checkmark}$ [1.2 min]	$\overset{5}{\checkmark}$ [1.8 s]
	7	$\overset{6}{\checkmark}$ [< 1 s]	$\overset{6}{\checkmark}$ [6.0 ms]	$\overset{6}{\checkmark}$ [12 min]	$\overset{?_6}{\checkmark}$ [OoM]	$\overset{6}{\checkmark}$ [< 1 s]	$\overset{6}{\checkmark}$ [2.0 ms]	$\overset{6}{\checkmark}$ [1.1 h]	$\overset{6}{\checkmark}$ [7.0 s]

Comparison over the four tools. Among the nine benchmarks, IronMask fails on one under standard probing model and fails on two under glitch-extended probing model. All successfully verified benchmarks are completed within one second (the timing reported by IronMask is less precise than that of the other tools). Notably, IronMask is able to verify ISW multiplication with 6 shares but encounters a segmentation fault when checking the multiplication with the smallest number of shares. Despite claiming completeness in the presence of glitches, IronMask does not uphold this claim. IronMask reports the ISW multiplication as 1-NI in the glitch-extended probing model for cases where the number of input shares is odd: in addition to the 3-shared and 5-shared ISW multiplications listed in Table 8, it also reports the 7-shared and 9-shared ISW multiplications as 1-NI. The results are reproducible, and the verification log is public available[6]. In contrast, maskVerif generally demonstrates the best performance under both models.

On the other hand, Prover does not surpass SILVER in verifying the standard probing security of higher-order standard gadgets. This discrepancy arises because the benchmarks provided by IronMask inadvertently equips SILVER with the variable ordering 1 strategy proposed in Section 4, rather than the original inefficient ordering. Meanwhile, Prover uses the default ordering 2 strategy, which is less suitable for verifying standard gadgets. Consequently, the suboptimal ordering 2 strategy leads to Prover running out of memory. Under the glitch-extended probing model, Prover generally performs better than SILVER

[6] https://github.com/Lucien98/IronMask/tree/main/experiment/im_results/ISW_mult

due to the application of reduction rules.

5.5 Analysis on the Insecure Implementations

There are two benchmarks that are not probing secure under both models. The first one, TINU_1 implementation of PRESENT S-box, is available at SILVER's repository. In [EGMP17], the PRESENT S-box is decomposed as $\mathcal{F} \circ \mathcal{G}$. The authors introduced correction terms to ensure the output sharing of $\mathcal{G}$ is uniform, resulting in the design TIU_1. However, the TINU_1 implementation of PRESENT S-box uses the function $\mathcal{G}$ without correction terms, thereby exhibiting leakage. More details about the leakage could be found at Prover's Github repository.

To our knowledge, this paper is the first one to identify the security issue of the L_2^4 implementation of PRINCE S-box from [BDMS22]. Both SILVER and Prover detected a second-order leakage for this implementation under both models (maskVerif identified the real leakage under standard probing model). For detailed expressions of the L_2^4 implementation of PRINCE S-box, please refer to Appendix F in [BDMS22]. In the following, the inputs of this S-box are denoted as a, b, c, d while $a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2, d_0, d_1, d_2$ are the corresponding shares. Under standard probing model, a probe placed at the partial result of h_8 in $\mathcal{F}$ function, $P_1 = a_1c_0 + c_0d_1$, combining a probe at the partial results of k_6 in $\mathcal{G}$ function, $P_2 = d_0^g b_0^g + d_0^g c_0^g + b_0^g + a_0^g$, results in second-order leakage. The variables can be expressed in terms of shares of a, b, c, d as shown in Equation 8 where the variables with a superscript f or g come from function $\mathcal{F}$ or $\mathcal{G}$.

$$
\begin{aligned}
&P_1 = a_1c_0 + c_0d_1 \\
&a_0^g = x_0^f = 1 + a_1 + d_1 + c_2 \\
&b_0^g = y_0^f = c_1 \\
&c_0^g = z_0^f = 1 + a_1 + a_1c_1 + c_1d_1 + c_2 + c_2a_1 + c_2d_1 + c_1a_2 + c_1d_2 + r_0 + r_2 \\
&d_0^g = t_0^f = d_1a_1 + 1 + d_1a_2 + b_1 + a_1d_2 + r_6 + r_8
\end{aligned}
\tag{8}
$$

Here, r_0 is only used by c_0^g, r_6 is only used by d_0^g. According to the theory of maskVerif, we could substitute c_0^g and d_0^g with r_0 and r_6 respectively in the expression of P_2. Therefore, the observable set $\{P_1, P_2\}$ is equivalent to $\{P_1', P_2'\} = \{c_0(a_1 + d_1), r_6c_1 + r_6r_0 + c_1 + (1 + a_1 + d_1 + c_2)\}$. The histogram of the joint distribution of (P_1', P_2') for $c = 0$ and $c = 1$ is depicted in Table 9. The distribution is dependent on the value of c, indicating leakage.

Table 9: Histogram of the joint distribution of (P_1', P_2') for $c = 0$ and $c = 1$

(P_1', P_2')	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$c = 0$	28	20	4	12
$c = 1$	20	28	12	4

One can observe P_1 includes one share of c in its expression while P_2 involves another two shares of c (in the expression of a_0^g and b_0^g), resulting the leakage. One might question whether changing the computation order of h_8 and k_6 could mitigate this issue. For example, if we use the mask o_0 from the expression of k_6 to mask P_2, P_2 would have a perfect mask o_0. Consequently, when combined with P_1, it will not reveal information about c. More specifically, we reorder the computation of k_6 as follows: first compute $P_3 = o_0 + a_0^g$, then incorporate the other terms into P_3 to obtain k_6. We manually adjusted the netlist generated by Design Compiler in this manner and tested it with SILVER and Prover, both of which reported no second-order leakage for the modified netlist file under the standard probing model. However, second-order leakage persisted under the glitch-extended probing model.

Under glitch-extended probing model, placing a probe at the input of the register z'_8 in function $\mathcal{F}$ would expose c_0, a_1, d_1. Another probe placed at k_6 in function $\mathcal{G}$ would reveal $a_0^g = 1 + a_1 + d_1 + c_2$ and $b_0^g = c_1$. It is evident that c can be reconstructed by combining these variables. To address this issue, we modified the design by replacing the correction term c_2 in the expression of x_0^f and x_1^f in function $\mathcal{F}$ with a fresh random bit r. This revised design is referred to as $\mathrm{L}_{2'}^4$ implementation of PRINCE S-box in Table 2. Its security is successfully verified by Prover within 13 minutes, whereas SILVER runs out of time and other tools encounter false positive issues.

6 Conclusion

In this study, drawing inspiration from auxiliary data structures from [EWS14] and the OPT rule from maskVerif [BBD+15], we introduced two reduction rules and two variable ordering strategies to enhance SILVER's capability in verifying standard and glitch-extended probing security, as well as uniformity of masked implementations. This effort led to the development of a tool named Prover. Thanks to these reduction rules, observation sets and secret input sets are significantly decreased before being verified by ROBDDs or even becoming empty sets. This reduction mitigates the reliance for ROBDDs and greatly enhances scalability. We also conducted numerous experiments to compare Prover to state-of-the-art tools, CocoAlma, maskVerif, and SILVER in verifying S-boxes. On verifying true negatives of existing efficient tool CocoAlma and maskVerif, Prover generally performs comparably or better than CocoAlma, although it is not as efficient as maskVerif. When verifying false positives of CocoAlma and maskVerif, Prover outperforms SILVER across most benchmarks. Notably, Prover successfully verified a design that SILVER could not complete within time limit, while other tools encountered false positive issues. While Prover's performance is not entirely satisfactory in verifying standard gadgets under standard probing model, it can handle verification at orders up to 4 or 5, which is sufficient for practical security needs. Additionally, our experiments helped identify bugs in existing tools, including SILVER and IronMask. In summary, Prover achieves a superior balance between efficiency and accuracy than the other state-of-the-art tools.

Future research directions could involve extending our methods to verify implementations under other security notions (such as NI, SNI, and PINI) and against other physical attacks, such as transitions.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 62172395), the Natural Science Foundation of Beijing - Innovation Joint Fund of Changping (No. L234085), and the Joint Funds of the National Natural Science Foundation of China (No. U2336210).
The authors sincerely thank the anonymous reviewers for their valuable comments, whichsignificantly improved the quality of the paper.

References

[Ajt11] Miklós Ajtai. Secure computation with information leaking to an adversary. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 715–724. ACM Press, June 2011.

[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, Heidelberg, 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, 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.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, Heidelberg, August 2016.

[BDMS22] Tim Beyne, Siemen Dhooghe, Amir Moradi, and Aein Rezaei Shahmirzadi. Cryptanalysis of efficient masked ciphers: Applications to low latency. *IACR TCHES*, 2022(1):679–721, 2022.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 321–353. Springer, Heidelberg, April / May 2018.

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight private circuits: Achieving probing security with the least refreshing. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 343–372. Springer, Heidelberg, December 2018.

[Bil15] Begül Bilgin. *Threshold implementations : as countermeasure against higher-order differential power analysis*. PhD thesis, University of Twente, Enschede, Netherlands, 2015.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, Heidelberg, August 2016.

[BJK+20] Christof Beierle, Jeremy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. SKINNY-AEAD and SKINNY-hash. *IACR Trans. Symm. Cryptol.*, 2020(S1):88–131, 2020.

[BMRT22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. IronMask: Versatile verification of masking security. In *2022 IEEE Symposium on Security and Privacy*, pages 142–160. IEEE Computer Society Press, May 2022.

[BNN+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small s-boxes. *Cryptogr. Commun.*, 7(1):3–33, 2015.

[BRNI13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 293–310. Springer, Heidelberg, August 2013.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[CGLS20] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. Cryptology ePrint Archive, Report 2020/185, 2020. https://eprint.iacr.org/2020/185.

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

[DBR19] Lauren De Meyer, Begül Bilgin, and Oscar Reparaz. Consolidating security notions in hardware masking. *IACR TCHES*, 2019(3):119–147, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8291.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, Heidelberg, May 2014.

[DRB+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 194–212. Springer, Heidelberg, August 2016.

[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, Heidelberg, December 2017.

[EWS14] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR TCHES*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270.

[Fri92] Joel Friedman. On the bit extraction problem. In *33rd FOCS*, pages 314–319. IEEE Computer Society Press, October 1992.

[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 Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 95–112. Springer, Heidelberg, February 2017.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of KECCAK. In Hana Kubátová, Martin Novotný, and Amund Skavhaug, editors, *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, pages 205–212. IEEE Computer Society, 2017.

[GXSC21] Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. A hybrid approach to formal verification of higher-order masked arithmetic programs. *ACM Trans. Softw. Eng. Methodol.*, 30(3):26:1–26:42, 2021.

[HB21] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 1–10. IEEE, 2021.

[HSSW10] Kevin Henshall, Peter Schachte, Harald Søndergaard, and Leigh Whiting. An algorithm for affine approximation of binary decision diagrams. *Chic. J. Theor. Comput. Sci.*, 2010, 2010.

[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, 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, Heidelberg, August 1999.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1799–1812. ACM Press, November 2022.

[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, 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, Heidelberg, December 2020.

[MCS22] Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert. Handcrafting: Improving automated masking in hardware with manual optimizations. Cryptology ePrint Archive, Report 2022/252, 2022. https://eprint.iacr.org/2022/252.

[Mil98] D. Michael Miller. An improved method for computing a generalized spectral coefficient. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 17(3):233–238, 1998.

[MKSM22] Nicolai Müller, David Knichel, Pascal Sasdrich, and Amir Moradi. Transitional leakage in theory and practice unveiling security flaws in masked circuits. *IACR TCHES*, 2022(2):266–288, 2022.

[MOPT12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 58–75. Springer, Heidelberg, September 2012.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, Heidelberg, February 2005.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, Heidelberg, August / September 2005.

[MS16] Amir Moradi and Tobias Schneider. Side-channel analysis protection and low-latency in action – case study of PRINCE and Midori –. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 517–547. Springer, Heidelberg, December 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, Heidelberg, December 2006.

[OMHE17] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Symbolic approach for side-channel resistance analysis of masked assembly codes. In Ulrich Kühne, Jean-Luc Danger, and Sylvain Guilley, editors, *PROOFS 2017, 6th International Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, September 29th, 2017*, volume 49 of *EPiC Series in Computing*, pages 17–32. EasyChair, 2017.

[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.

[RBN+15a] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 764–783. Springer, Heidelberg, August 2015.

[RBN+15b] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. Cryptology ePrint Archive, Report 2015/719, 2015. https://eprint.iacr.org/2015/719.

[SDM23] Aein Rezaei Shahmirzadi, Siemen Dhooghe, and Amir Moradi. Low-latency and low-randomness second-order masked cubic functions. *IACR TCHES*, 2023(1):113–152, 2023.

[SM21a] Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes. *IACR TCHES*, 2021(1):305–342, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8736.

[SM21b] Aein Rezaei Shahmirzadi and Amir Moradi. Second-order SCA security with almost no fresh randomness. *IACR TCHES*, 2021(3):708–755, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8990.

[Sny04] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, volume 79, pages 122–148, 2004.

[TN95] Mitchell A. Thornton and V. S. S. Nair. Efficient calculation of spectral coefficients and their applications. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 14(11):1328–1341, 1995.

[Tri03] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003. https://eprint.iacr.org/2003/236.

[UHA17] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward more efficient DPA-resistant AES hardware architecture based on threshold implementation. In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 50–64. Springer, Heidelberg, April 2017.

[WGK13] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

[YCW+24] Fu Yao, Hua Chen, Yongzhuang Wei, Enes Pasalic, Feng Zhou, and Limin Fan. Optimizing AES threshold implementation under the glitch-extended probing model. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 43(7):1984–1997, 2024.

[ZGSW18] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. Scinfer: Refinement-based verification of software countermeasures against side-channel attacks. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 157–177. Springer, 2018.

[ZSS+21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziyeh Salarifard, and Amir Moradi. Low-latency keccak at any arbitrary order. *IACR TCHES*, 2021(4):388–411, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9070.