IronMask: Versatile Verification of Masking Security

Sonia Belaïd[1], Darius Mercadier[1], Matthieu Rivain[1], and Abdul Rahman Taleb[1,2]

[1] CryptoExperts, France
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

{sonia.belaid,darius.mercadier,matthieu.rivain,abdul.taleb}@cryptoexperts.com

Abstract. This paper introduces IronMask, a new versatile verification tool for masking security. IronMask is the first to offer the verification of standard simulation-based security notions in the probing model as well as recent composition and expandability notions in the random probing model. It supports any masking gadgets with linear randomness (*e.g.* addition, copy and refresh gadgets) as well as quadratic gadgets (*e.g.* multiplication gadgets) that might include non-linear randomness (*e.g.* by refreshing their inputs), while providing complete verification results for both types of gadgets. We achieve this complete verifiability by introducing a new algebraic characterization for such quadratic gadgets and exhibiting a complete method to determine the sets of input shares which are necessary and sufficient to perform a perfect simulation of any set of probes. We report various benchmarks which show that IronMask is competitive with state-of-the-art verification tools in the probing model (maskVerif, scVerif, SILVER, matverif). IronMask is also several orders of magnitude faster than VRAPS –the only previous tool verifying random probing composability and expandability– as well as SILVER –the only previous tool providing complete verification for quadratic gadgets with non-linear randomness. Thanks to this completeness and increased performance, we obtain better bounds for the tolerated leakage probability of state-of-the-art random probing secure compilers.

Keywords: Side-channel security, masking, physical defaults, automatic verification, complete verification, composition, probing model, random probing model, IronMask

1 Introduction

Side-channel attacks exploit the physical leakage of a device executing cryptographic implementations to extract the manipulated secrets. They can be built from cheap equipment and are generally able to recover the keys in a limited time in the absence of specific protections.

Since the discovery of these attacks in the late nineties, the community investigated several approaches to counteract them. Among these approaches, *masking* [22,30] is one of the most deployed in practice. The main idea of masking is to split the information between several variables called *shares*. In a nutshell, for t^{th}-order Boolean masking, each sensitive value x is split into $(t+1)$ shares $x_1, \ldots, x_{t+1}$. The first t shares $x_1, \ldots, x_t$ are generated uniformly at random and the last one x_{t+1} is computed as $x_{t+1} \leftarrow x \oplus x_1 \oplus \cdots \oplus x_t$. Doing so, the attacker must collect and then aggregate the information from all the shares to recover sensitive data. This becomes exponentially hard with the number of shares as each observation comes with noise.

While Boolean masking can be easily implemented for linear operations that are directly applied on each share separately, it becomes much more complex for non-linear computations for which shares must be interleaved. These operations require additional randomness to preserve the security order.

To properly reason on the security of these operations, and more generally on the security of masked implementations, the leakage is generally formalized in so-called leakage models. The most famous one is probably the t-probing model, introduced by Ishai, Sahai, and Wagner in 2003 [32].

In this model, the leakage is modeled as the exact values of t intermediate variables chosen by the attacker. Variants of this model include the *robust probing model* [28] in hardware scenarios, which considers wider leakage to model physical effects such as glitches. Instead of t intermediate variables, the attacker gets t sets of variables that belong to the same combinatorial set. While probing-like models are very convenient to build security proofs, they sometimes fail to closely reflect the reality of embedded devices. For instance, they do not capture *horizontal attacks* [9], which exploit in particular the repeated manipulation of variables within an execution.

Therefore, the community is starting to focus on more realistic leakage models, such as the *random probing model* [1,26]. The leakage is assumed to gather the exact value carried out by each wire of the circuit with probability p. The security tightly reduces to the security in the *noisy leakage model* [35,26] in which each variable leaks a noisy function of its value. The random probing model can also be extended to capture glitches or other side effects.

In any model, the security of almost all masking approaches relies on the security of small gadgets (*i.e.*, atomic masked operations) and their composition. For instance, in the probing model and its variants, the type-system of Barthe *et al.* [6] makes it possible to securely compose small gadgets that are proven to be (strong) non-interferent. The t-non-interference is itself satisfied when any set of t intermediate variables can be perfectly simulated from t input shares while the t-strong non-interference adds a condition on the output observations. Both properties imply t-probing security. Similarly, in the random probing model, Ananth *et al.* [2] use an expansion strategy on top of secure multi-party computation protocols while Belaïd *et al.* use an expansion strategy on top of random probing expandable gadgets [13,16,17]. The latter works only require gadgets that are masked at small orders to build circuits achieving an arbitrary level of random probing security.

Even at reasonable orders, the manual verification of security properties on small gadgets has been shown to be very error-prone [24]. Therefore, automatic tools are regularly built to apply a formal verification. For instance, two existing tools currently check random probing properties (namely, VRAPS [13] and STRAPS [20]). However, none of these two tools provide *completeness* (*i.e.*, they can falsely deem a set of leaking variables as insecure with respect to the secret). In terms of efficiency, VRAPS makes it difficult to verify even small gadgets at reasonable orders and STRAPS only manages to do it using verification rules from the underlying tool maskVerif [5,6,3] (which only verifies probing security properties), which by construction, are not complete.

As for tools which achieve complete verification of some (probing like) security properties, we count only SILVER [33], which suffers from low performance, and matverif [19], which is restricted to specific gadgets only.

Our contributions. In this work, we introduce IronMask, a new automatic verification tool to check probing and random probing security properties using complete and efficient procedures. Our main contributions are the following:

– We formalize all of the probing and random probing properties of the state of the art from a single common building block, a function we call SIS, and show that all of the security properties can be verified using a unique instantiation of that function (Section 2). In a nutshell, SIS receives as input a set of probes on a gadget (and the description of the corresponding gadget) and performs a number of operations on the algebraic expressions of the probes in order to determine the exact sets of input shares which are necessary and sufficient to perform a perfect simulation of these probes. While SIS partially uses some properties from the state of the art,

it was not clearly exhibited before, and the unification of all the (random) probing-like security notions with respect to this function was not explicitly well-defined.
- We extend the algebraic characterization introduced in [11,12] of gadgets with linear randomness (*i.e.* all random values are additive on the wires of the gadget) to more general gadgets with non-linear randomness which perform quadratic operations on input shares mixed with randomness (Section 3). Our extended characterization notably captures recent gadget designs [9,16], achieving advanced security properties such as resistance to *horizontal attacks* or random probing security. From this characterization, we provide a complete verification method that applies to most (if not all) masking gadgets for standard operations (addition, multiplication, refreshing, etc.). In comparison, the only previously existing complete verification method for such general gadgets would rely on exhausting the truth table of tuples of intermediate variables with respect to the inputs and the randomness, which is highly inefficient.
- We introduce IronMask, a new versatile verification tool for all probing and random probing-like properties in the state of the art (Section 4). IronMask supports the verification of traditional gadgets with linear randomness, as well as newly formalized gadgets with non-linear randomness along with a complete verification method for both types of gadgets based on our extended algebraic characterization. IronMask implements several optimizations to make the verification faster. We benchmark the performance of our new tool (Section 5) and show that it is competitive with state-of-the-art verification tools in the probing model (maskVerif, scVerif, SILVER, matverif). IronMask is also several orders of magnitude faster than VRAPS, the only previous tool verifying random probing composability and expandability, and SILVER, the only previous tool providing complete verification for gadgets with non-linear randomness. IronMask uses complete methods for the verification, unlike VRAPS and STRAPS which are the only verification tools in the random probing model. Thanks to this completeness and increased performance, we obtain better bounds for the tolerated leakage probability[3] of state-of-the-art random probing secure compilers. IronMask is open-source and publicly available at:

`https://github.com/CryptoExperts/IronMask`

2 Characterization of Security Notions for Masking Gadgets

2.1 Preliminaries

Throughout the paper, $\mathbb{K}$ shall denote a finite field. For any $n \in \mathbb{N}$, we shall denote $[n]$ the integer set $[n] = [1, n] \cap \mathbb{Z}$. For any tuple $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{K}^n$ and any set $I \subseteq [n]$, we shall denote $\vec{x}|_I = (x_i)_{i \in I}$.

An *arithmetic circuit* over a field $\mathbb{K}$ is a labeled directed acyclic graph whose edges are *wires* and vertices are *arithmetic gates* processing operations over $\mathbb{K}$ (*e.g.*, additions, multiplications, copies). A *randomized arithmetic circuit* is equipped with an additional random gate of fan-in 0 which outputs a fresh uniform random value of $\mathbb{K}$.

In the following, the *n-linear decoding* mapping, denoted LinDec, refers to the function $\bigcup_n \mathbb{K}^n \to \mathbb{K}$ defined as

$$\mathsf{LinDec} : (x_1, \ldots, x_n) \mapsto x_1 + \cdots + x_n \ ,$$

[3] Increased performance yields a tighter approximation of the *failure* probability in the random probing model, while completeness allows to compute more accurate and complete values for that probability. Both factors result in a tighter, more accurate approximation of the failure probability, from which tighter bounds on the tolerated leakage probability are derived. See Section 5.1 for more details.

for every $n \in \mathbb{N}$ and $(x_1, \ldots, x_n) \in \mathbb{K}^n$. We shall further consider that, for every $n, \ell \in \mathbb{N}$, on input $(\vec{x}_1, \ldots, \vec{x}_\ell) \in (\mathbb{K}^n)^\ell$ the n-linear decoding mapping acts as

$$\mathsf{LinDec} : (\vec{x}_1, \ldots, \vec{x}_\ell) \mapsto (\mathsf{LinDec}(\vec{x}_1), \ldots, \mathsf{LinDec}(\vec{x}_\ell)) \ .$$

For any $x \in \mathbb{K}$, an *n-linear sharing* of x is a vector $\vec{x} \in \mathbb{K}^n$ such that $\mathsf{LinDec}(\vec{x}) = x$. A random vector $\vec{x}$ distributed over $\mathbb{K}^n$ is said to be a *uniform sharing* of $x := \mathsf{LinDec}(\vec{x})$ if for any set $I \subseteq [n]$ with $|I| < n$ the random vector $\vec{x}|_I$ is uniformly distributed over $\mathbb{K}^{|I|}$.

In the following, we shall call an (*n-share*, *ℓ-to-m*) *gadget* denoted G for some function $g : \mathbb{K}^\ell \to \mathbb{K}^m$, a randomized arithmetic circuit that maps an input $(\vec{x}_1, \ldots, \vec{x}_\ell) \in (\mathbb{K}^n)^\ell$ to an output $(\vec{y}_1, \ldots, \vec{y}_m) \in (\mathbb{K}^n)^m$ such that

$$\big(\mathsf{LinDec}(\vec{y}_i)\big)_{1 \le i \le m} = g\Big[\big(\mathsf{LinDec}(\vec{x}_i)\big)_{1 \le i \le \ell}\Big]$$

with probability 1 over the internal randomness of G.

For any wire on a randomized arithmetic circuit, we shall call a *probe* on this wire, the symbolic expression of the circuit inputs (shares) and the generated randoms (outputs of random gates) associated to this wire.

In the following, we shall say that a tuple $\vec{P}$ of symbolic expressions of input sharings $\vec{x}_1$, …, $\vec{x}_\ell$ (of size n) and randomness $\vec{r}$ (of size ρ) can be *perfectly simulated* from $\vec{x}_1|_{I_1}$, …, $\vec{x}_\ell|_{I_\ell}$ if and only if for any distributions $\mathcal{D}_{\vec{x}_1}$, …, $\mathcal{D}_{\vec{x}_\ell}$ over $\mathbb{K}^n$ there exists a probabilistic algorithm $\mathcal{S}$ (the simulator) such that given $\vec{x}_1 \leftarrow \mathcal{D}_{\vec{x}_1}$, …, $\vec{x}_\ell \leftarrow \mathcal{D}_{\vec{x}_\ell}$, $\vec{r} \leftarrow \mathbb{K}^\rho$, we have

$$\mathcal{S}(\vec{x}_1|_{I_1}, \ldots, \vec{x}_\ell|_{I_\ell}) \overset{\text{id}}{=} \mathsf{eval}_{\vec{P}}(\vec{x}_1, \ldots, \vec{x}_\ell, \vec{r}) \ , \tag{1}$$

where $\leftarrow$ means randomly sampling from a set or a distribution, $\overset{\text{id}}{=}$ means identically distributed and $\mathsf{eval}_{\vec{P}}$ is the function which outputs an evaluation of the symbolic expressions in $\vec{P}$ from its arguments.

2.2 Security Notions

To be composed in secure circuits, gadgets are generally expected to satisfy slightly stronger security notions than (random) probing security[4]. In a nutshell, security proofs generally demonstrate that the observations made on a gadget and/or on its output shares can be perfectly simulated from a subset of its input shares. If this subset is strictly smaller than the set of input shares and if the sharing is uniform (which is the case in tight masking circuits), then the observations can be simulated independently from the secrets and are therefore independent from the secret. Such properties also make it possible to compose secure gadgets by analyzing the propagation of the simulated observations.

Although many different security notions have been introduced to build proofs of gadgets in the (random) probing model, we show that they can almost all be defined on top of a single building block: the *set of input shares* (SIS) function. The latter takes as input a set of probes on internal wires of the gadget as well as a set of output shares, and returns a set of input shares necessary (and sufficient) to perfectly simulate these internal probes and output shares. We formalize the SIS

[4] Although some composition rules were established directly between probing secure gadgets (*e.g.*, [15]), their scope remains limited.

primitive in Definition 1 before showing how to use it to express state-of-the-art properties. For the sake of clarity, we restrain the following definitions to the case of single output gadgets (which is the most common case), but the extension to multi-output gadgets is straightforward. We denote SIS_G to be SIS with input gadget G.

Definition 1. *Let G be an (n-share, ℓ-to-1) gadget mapping ℓ input sharings $(\vec{x}_1, \ldots, \vec{x}_\ell) \in (\mathbb{K}^n)^\ell$ to an output sharing $\vec{y} \in \mathbb{K}^n$. Let $\vec{P}$ be a set of probes on G and $O \subseteq [n]$ a set of output indices. The function SIS_G maps $\vec{P}$ and O to the unique smallest sets of input indices $I_1, \ldots, I_\ell$ such that $(\vec{P}, \vec{y}|_O)$ can be perfectly simulated from $\vec{x}_1|_{I_1}, \ldots, \vec{x}_\ell|_{I_\ell}$.*

Note that for any gadget G, the *smallest* set of input shares returned by SIS_G is *uniquely* defined from the result [11, Lemma 7.5], which demonstrates that if a set of probes can be simulated from different sets of inputs shares, then it can also be simulated by the intersection of these sets.

Probing security notions. We now formalize the *probing-like* security notions (*i.e.*, to achieve security and secure composition in the probing model) for any n-share, ℓ-to-1 gadget G (all these notions can be generalized for the case of multiple outputs). Definition 2 recalls the non-interference (NI) property from [6].

Definition 2 (t-NI). *A gadget G is t-NI if for any tuple $\vec{P}$ of t_1 internal probes and any set O of t_2 output share indices such that $t_1 + t_2 \leq t$, the sets $(I_1, \ldots I_\ell) := \mathsf{SIS}_G(\vec{P}, O)$ satisfy $|I_i| \leq t, \ \forall i \in [\ell]$.*

Other common probing-like properties can be defined in a similar way by changing the condition on the sets $(I_1, \ldots, I_\ell)$ in the output of the SIS_G primitive. We list these conditions in Table 1 for some of the most common probing-like properties, with respect to t and O. While most of the properties are interesting in the context of composing secure gadgets to achieve global security, directly verifying the probing security of a complete implementation is useful in some cases, such as analyzing a complete 2-share AES implementation. To represent this case, we denote PS* the SIS-based probing security definition. PS* is actually equivalent to the case of $(n-1)$-NI with $O = \emptyset$ and $t = n - 1$. We list this property in Table 1 as well.

Table 1: Probing-like security notions from the basic primitive SIS_G. t_1 indicates the number of probes on input and intermediate variables, while t_2 indicates the number of probes on output shares, with $t = t_1 + t_2$.

Notion	Condition
t-NI [5]	$\lvert I_i\rvert \leq t, \ \forall i \in [\ell]$
t-SNI [6]	$\lvert I_i\rvert \leq t_1, \ \forall i \in [\ell]$
t-TNI [11]	$\lvert I_i\rvert \leq t_1 + t_2, \ \forall i \in [\ell]$
(t, f)-NI [4]	$\lvert I_i\rvert \leq f(t_1, t_2), \ \forall i \in [\ell]$
t-PINI [21]	$\lvert(\cup_i I_i) \setminus O\rvert \leq t_1$
PS*	$\lvert I_i\rvert \leq n - 1, \ \forall i \in [\ell]$

Because the probing model is often criticized as it fails to reflect the reality of embedded systems (see for instance *horizontal attacks* [9]), the community recently started to study the random probing model. Despite its complexity compared to the simple probing model, the random probing model was shown to be closer to the more realistic noisy leakage model, reducing the gap between theoretical proofs and concrete observations.

Random probing security notions. To formalize security notions in the random probing model, we rely on the LeakingWires procedure formalized in [13]. This (probabilistic) procedure outputs a tuple of probes $\vec{P}$ on the gadget G such that each wire of G is added to $\vec{P}$ independently with probability p, where p is the *leakage probability* (a.k.a. *leakage rate*). Definition 3 recalls the *random probing composability* (RPC) notion [13] based on the SIS primitive.

Definition 3. *Let $p, \varepsilon \in [0,1]$ and $n, t \in [0,n]$. Let G be a n-share ℓ-to-1 gadget and let $\vec{P}$ be the random vector defined as $\vec{P} = \mathsf{LeakingWires}(G,p)$. Then G is (t,p,ε)-RPC if for every $O \subseteq [n]$ with $|O| = t$, the sets $(I_1,\ldots,I_\ell) = \mathsf{SIS}_G(\vec{P},O)$ satisfy $\Pr[(|I_1| > t) \vee \ldots \vee (|I_\ell| > t)] \leq \varepsilon$, where the probability is taken over all tuples of probes $\vec{P}$ obtained through $\mathsf{LeakingWires}(G,p)$.*

We call the event $\big((|I_1| > t) \vee \ldots \vee (|I_\ell| > t)\big)$ a *failure event* (failure of a perfect simulation) and ε is the *failure probability* or the probability of a failure event to occur. The *random probing expandability* (RPE) notions introduced and analyzed in [13,16,17] can also be defined in a similar fashion. Like the authors of [13,16,17], we restrict its definition to 2-input circuits for the sake of clarity but recall that the extension is straightforward. We have that G is (t,p,ε)-RPE1 (resp. RPE2) if for every $O \subseteq [n]$ with $|O| = t$ (resp. if there exists $O \subseteq [n]$ with $|O| = n-1$), the sets $(I_1, I_2) = \mathsf{SIS}_G(\vec{P},O)$ satisfy

$$\big(\Pr[|I_2| > t] \leq \varepsilon\big) \ \wedge \ \big(\Pr[|I_2| > t] \leq \varepsilon\big) \wedge \ \big(\Pr[(|I_1| > t) \wedge (|I_2| > t)] \leq \varepsilon^2\big) \ .$$

The three random probing notions are summarized in Table 2. As in the probing security case earlier, it can be useful to directly verify the random probing security of a complete implementation. To represent this case, we denote (p,ε)-RPS* the SIS-based definition of random probing security. This notion is actually similar to the RPC definition, except that we do not consider probes on the outputs, *i.e.* $O = \emptyset$, and a failure occurs when all the shares (of one input) are necessary to perfectly simulate the probes, *i.e.* the failure event is

$$\Pr[(|I_1| = n) \vee \ldots \vee (|I_\ell| = n)] \leq \varepsilon \ .$$

In [13], the authors introduce a method to verify random probing properties by computing the

Table 2: Random probing-like security notions from the basic primitive SIS_G.

Notion	Output O	Condition(s)
(t,p,ε)-RPC [13]	$\forall\, O, \lvert O\rvert = t$	$\Pr[(\lvert I_1\rvert > t) \vee \ldots \vee (\lvert I_\ell\rvert > t)] \leq \varepsilon$
(t,p,ε)-RPE1 [13]	$\forall\, O, \lvert O\rvert = t$	$\big(\forall i,\ \Pr[(\lvert I_i\rvert > t)] \leq \varepsilon\big) \wedge \big(\Pr[(\lvert I_1\rvert > t) \wedge (\lvert I_2\rvert > t)] \leq \varepsilon^2\big)$
(t,p,ε)-RPE2 [13]	$\exists O, \lvert O\rvert = n-1$	$\big(\forall i,\ \Pr[(\lvert I_i\rvert > t)] \leq \varepsilon\big) \wedge \big(\Pr[(\lvert I_1\rvert > t) \wedge (\lvert I_2\rvert > t)] \leq \varepsilon^2\big)$
(p,ε)-RPS*	$O = \emptyset$	$\Pr[(\lvert I_1\rvert = n) \vee \ldots \vee (\lvert I_\ell\rvert = n)] \leq \varepsilon$

failure probability ε as a function $f(p)$ of the leakage probability p. For (p,ε)-RPS* of an n-share gadget of s wires for example, $\varepsilon = f(p)$ is computed as

$$f(p) = \sum_{\substack{\vec{P} \text{ s.t. } (I_1,\ldots,I_\ell)=\mathsf{SIS}(\vec{P},\emptyset) \\ |I_1|=n \ \vee \ldots \vee \ |I_\ell|=n}} p^{|\vec{P}|}(1-p)^{s-|\vec{P}|} \ . \tag{2}$$

In the above equation, we consider that each tuple of probes $\vec{P}$ on a gadget can exactly leak with probability $p^{|\vec{P}|}(1-p)^{s-|\vec{P}|}$ since each of the wires in $\vec{P}$ is added independently with probability p,

and each of the remaining wires does not leak with probability $1-p$. Then, out of all such possible tuples of wires, $f(p)$ represents the sum over the probabilities of obtaining tuples of probes only for which we get a failure event using SIS (the failure event being $(|I_1| = n \ \vee \ldots \vee \ |I_\ell| = n)$ in this context). For a gadget with a total of s wires, computing $f(p)$ then amounts to computing

$$f(p) = \sum_{i=1}^{s} c_i \ p^i (1-p)^{s-i} \tag{3}$$

where we simply group the probabilities with respect to the size of the tuples of probes. In other words, c_i is the number of tuples of i wires, for which we obtain a failure event using SIS. For instance, if there are exactly 2 tuples of probes $\vec{P_1}, \vec{P_2}$ for which we get a failure event and such that $|\vec{P_1}| = |\vec{P_2}| = 3$, then we get $c_3 = 2$ in equation (3). For other random probing properties, the computation is similar with the difference of considering the correct failure event with the correct t, and the condition on the output set of shares O which is not empty anymore. As shown in [13], a set of (t, p, ε)-RPE gadgets with $\varepsilon = f(p)$ can be expanded into a probing secure circuit (with arbitrary security level) for any leakage probability smaller than p_{max}, with $p_{max} \in [0, 1]$ being the solution of the equation $f(p) = p$. This threshold is called the *tolerated leakage probability* of the gadgets.

The recent Probe Distribution Table (PDT) of Cassiers *et al.* [20] can also be expressed in terms of our basic function SIS_G. The PDT is a two-dimensional table indexed by all possible sets of input indices $\vec{I} = (I_1, \ldots, I_\ell)$ where $I_i \subseteq [n]$ and by all possible sets of output indices $O \subseteq [n]$, defined as

$$\mathrm{PDT}[\vec{I}][O] := \sum_{\vec{P} \text{ s.t. } \vec{I} = \mathsf{SIS}(\vec{P}, O)} p^{|\vec{P}|}(1-p)^{s-|\vec{P}|} \tag{4}$$

where s is the number of wires in G. In other words, each entry in the PDT is a different function $f(p)$ as in equations (2), (3). Computing the PDT amounts to considering each possible tuple of probes $\vec{P}$ on the gadget, and compute $\mathsf{SIS}_G(\vec{P}, O) = \vec{I} = (I_1, \ldots, I_\ell)$ for each possible set of output shares indices $O \subseteq [n]$. Then, update the corresponding function in the PDT indexed by $\vec{I}$ and O as $\mathrm{PDT}[\vec{I}][O] = \mathrm{PDT}[\vec{I}][O] \ + \ p^{|\vec{P}|}(1-p)^{s-|\vec{P}|}$. When exploring all the possible sets of internal probes $\vec{P}$ and all the sets of output indices O, the output of SIS_G shall serve as a basis to compute the expected distributions.

We showed how standard probing and random probing security notions can be expressed in terms of the SIS_G function. In the next section, we focus on the algebraic characterization of masking gadgets and the concrete evaluation of the SIS_G function.

3 Algebraic Characterization of Masking Gadgets

In this section, we recall and extend the algebraic characterization of masking gadgets, and the subsequent security results. Previous works [11, 12] considered gadgets with linear randomness, *i.e.* all random values are additive on the wires of the gadget. We refer to these gadgets as LR-gadgets (see for instance multiplications and refresh gadgets from [32]). In this work, we extend the characterization to gadgets with non-linear randomness, *i.e.* on gadget performing non-linear operations on input shares mixed with randomness. We denote these gadgets as NLR-gadgets. Our extended characterization notably captures recent gadget designs, see *e.g.* [9, 16], achieving

advanced security properties such as resistance to horizontal attacks or random probing security. We also show how to verify the security of masking gadgets using this algebraic characterization by a concrete evaluation of the SIS primitive, which will be the core primitive of IronMask.

3.1 Characterization of Gadgets with Linear Randomness

In this paper, we call an LR-gadget any ℓ-to-m gadget $G : (\vec{x}_1, \ldots, \vec{x}_\ell) \mapsto (\vec{y}_1, \ldots, \vec{y}_m)$ with the output of the form:

$$(\vec{y}_1, \ldots, \vec{y}_m) := R\big(F(\vec{x}_1, \ldots, \vec{x}_\ell),\, \vec{r}\big) \ ,$$

where F is any arithmetic circuit, R is a linear arithmetic circuit (*i.e.* computing a linear function) and $\vec{r}$ is a vector of internal randomness uniformly drawn from $\mathbb{K}^\rho$. Formally, each coordinate of $\vec{r}$ is the output of a randomness gate of G, and F and R are composed solely of operation gates. Note that this characterization is more general than the one from [11,12], which only considers quadratic circuits for F. We show hereafter that we can still obtain an efficient and complete evaluation of SIS for those gadgets, which in turn yields an efficient verification of the considered security notions.

By definition, any probe on an LR-gadget can be written as

$$p = f_p(\vec{x}_1, \ldots, \vec{x}_\ell) + \vec{r}^{\,T} \cdot \vec{s}_p \tag{5}$$

for some arithmetic function $f_p : (\mathbb{K}^n)^\ell \to \mathbb{K}$ and some constant vector $\vec{s}_p \in \mathbb{K}^\rho$.

Given a tuple of probes $\vec{P} = (p_1, \ldots, p_d)$ on the gadget G, we are interested in determining the set of input shares necessary for a perfect simulation of all probes in $\vec{P}$. In particular, if $\vec{P}$ can be simulated with at most $n-1$ of each input sharing, then we know that $\vec{P}$ is independent of the secret inputs. Belaïd *et al.* [11] showed how to use a Gaussian elimination technique in order to determine the simulatability of a tuple of probes for gadgets with linear randomness over the binary field. This technique was later extended to any finite field in [12]. We base the verification procedure for LR-gadgets in our tool on this technique.

We start by stating the result with Gaussian elimination from [11,12] in a different formulation that is more convenient for our purposes. For this, we first define a simple function shares(.), which takes as input a tuple of symbolic expressions $(e_1, \ldots, e_d)$ of the input shares, *i.e.* $e_i = f_{e_i}(\vec{x}_1, \ldots, \vec{x}_\ell)$ for some algebraic function f_{e_i}, and which outputs the (smallest) sets of indices I_1, ..., I_ℓ such that $(e_1, \ldots, e_d)$ functionally depends on $(\vec{x}_1|_{I_1}, \ldots, \vec{x}_\ell|_{I_\ell})$. Notice that evaluating shares(.) simply consists in extracting the indices of the input shares that are contained in the symbolic expressions $(e_1, \ldots, e_d)$. We stress that the input shares $(\vec{x}_1|_{I_1}, \ldots, \vec{x}_\ell|_{I_\ell})$ where $(I_1, \ldots, I_\ell) := \mathsf{shares}(e_1, \ldots, e_d)$ are necessary and sufficient for a perfect simulation of $(e_1, \ldots, e_d)$. Note that shares(.) is executed on the tuple of expressions that is the output tuple of the Gaussian elimination technique. In fact, after executing the Gaussian elimination, we are guaranteed that the remaining expressions cannot be simplified any further in the given field $\mathbb{K}$ and they are solely formed of operations between input shares (they do not include any random variables). In this case, to perfectly simulate the resulting tuple (which is equivalent to perfectly simulating the tuple given before Gaussian elimination), there is no choice but to have access to all of the input shares that are involved in the remaining expressions, which is why shares(.) simply extracts the indices of these input shares.

Lemma 1. *Let G be an n-share gadget. Let $\vec{P} = (p_1, \ldots, p_d)$ be a tuple of probes on G. Let $S \in \mathbb{K}^{d \times \rho}$ be the matrix such that*

$$S = (\vec{s}_{p_1} \mid \vec{s}_{p_2} \mid \cdots \mid \vec{s}_{p_d})$$

(i.e. each $\vec{s}_{p_i}$ is a column vector of S) and let S' be the row reduced form of the matrix S such that S' is of the form

$$S' = \begin{pmatrix} 0_{m,d-m} & 0_{m,R-d+m} \\ I_{d-m} & S'' \end{pmatrix}$$

up to some permutations on the rows with $S' = N \cdot S$ where N is an invertible matrix in $\mathbb{K}^{d\times d}$. Let $\vec{P}'$ be defined as

$$\vec{P}' = N \cdot \vec{P} = (p'_1, \ldots, p'_m, p'_{m+1}, \ldots, p'_d) \ .$$

Then, the set of input shares necessary to simulate the probes in $\vec{P}$ is $\mathsf{shares}(p'_1, \ldots, p'_m)$.

Sketch of proof. The proof of the result follows the proof of Theorem 3.1 from [11] and Theorem 3.2 of [12]. It is shown in the latter that we can perfectly simulate the probes in $\vec{P}$ by perfectly simulating all probes in $\vec{P}'$, since the matrix N is invertible and we can obtain $\vec{P}$ from $N^{-1}\cdot\vec{P}'$. Then, to perfectly simulate probes in the tuple $\vec{P}'$, we observe from S' that each algebraic expression in the tuple $(p'_{m+1}, \ldots, p'_d)$ contains a random value that does not appear in any other algebraic expression in $\vec{P}'$. We can thus perfectly simulate $(p'_{m+1}, \ldots, p'_d)$ by generating $d-m$ uniform random values without the need of any input shares. The remaining algebraic expressions $(p'_1, \ldots, p'_m)$ contain no random values and are all of the form $p'_i = f_{p'_i}(\vec{x}_1, \ldots, \vec{x}_m)$ for $i \in [m]$. Hence, to perfectly simulate each of them we need (and only need) the input shares which are involved in each $f_{p'_i}(\vec{x}_1, \ldots, \vec{x}_m)$, namely the inputs shares indexed by $(I_1, \ldots, I_\ell) := \mathsf{shares}(p'_1, \ldots, p'_m)$. Using the input shares $(\vec{x}_1|_{I_1}, \ldots, \vec{x}_\ell|_{I_\ell})$ we can perfectly simulate $(p'_1, \ldots, p'_m)$ and thus perfectly simulate all algebraic expressions in $\vec{P}'$, from which we get a perfect simulation of the probes in $\vec{P}$.

Lemma 1 actually provides a way to evaluate the function SIS in the case of LR-gadgets. Note that the set of probes $\vec{P}$ in the lemma must be defined as the union of $\vec{P}$ and $\vec{y}|_O$ in an evaluation of $\mathsf{SIS}_G(\vec{P}, O)$ (while used to define security notions, SIS is based on two arguments to differentiate probes on internal wires and probes on output shares whereas this distinction is not used in the evaluation process of Lemma 1). According to the above lemma, an evaluation of SIS simply consists of a row reduction on the matrix of the random dependencies (S), after which the function $\mathsf{shares}(.)$ is used on the obtained expressions without random values (*i.e.* $(p'_1, \ldots, p'_\ell)$ in the lemma). The output of SIS is then exactly the output of $\mathsf{shares}(.)$, which is the set of input shares necessary for a perfect simulation of all the probes. We show in Section 4 how this technique is efficiently implemented in our verification tool.

3.2 Characterization of Gadgets with Non-Linear Randomness

In this section, we extend the algebraic characterization for LR-gadgets of Section 3.1 to NLR-gadgets, *i.e.* gadgets performing non-linear operations on input shares mixed with randomness. An NLR-gadgets is an ℓ-to-m gadget $G : (\vec{x}_1, \ldots, \vec{x}_\ell) \mapsto (\vec{y}_1, \ldots, \vec{y}_m)$ with the output of the form:

$$(\vec{y}_1, \ldots, \vec{y}_m) := R_{\ell+1}\big(F(R_1(\vec{x}_1, \vec{r}_1), \ldots, R_\ell(\vec{x}_\ell, \vec{r}_\ell)), \vec{r}_{\ell+1}\big)$$

where F is any arithmetic circuit, the R_i are linear arithmetic circuits and the $\vec{r}_i$ are vectors of random values uniformly drawn from $\mathbb{K}^{\rho_i}$. We further assume that F computes a homogeneous multi-linear form, namely $F(\vec{z}_1, \ldots, \vec{z}_\ell)$ is a sum of degree-ℓ monomials, each of which being a product containing exactly one coordinate from each $\vec{z}_i$.

For the sake of clarity, we describe the verification method for the particular case of 2-input gadgets; the extension to ℓ inputs is straightforward. We thus present NLR-gadgets as 2-to-m gadgets $G : (\vec{x}_1, \vec{x}_2) \mapsto (\vec{y}_1, \ldots, \vec{y}_m)$ with the output of the form:

$$(\vec{y}_1, \ldots, \vec{y}_m) := R_3\big(F(R_1(\vec{x}_1, \vec{r}_1), R_2(\vec{x}_2, \vec{r}_2)), \vec{r}_3\big)$$

This characterization notably covers a wide majority (if not the totality) of multiplication gadgets. It covers in particular multiplication gadgets which first start by refreshing one of (resp. each of) their inputs before performing sharewise products that are finally recombined into the output sharing (with additional randomness). Such multiplication gadgets have been recently described in [31, 13, 16, 21].

Any probe on such an NLR-gadget is either a probe on the inner circuits $R_i(\vec{x}_i, \vec{r}_i)$ and is of the form:

$$p = \vec{x}_i^{\mathsf{T}} \cdot \vec{w}_p + \vec{r}_i^{\mathsf{T}} \cdot \vec{s}_p \tag{6}$$

for $i \in \{1, 2\}$ (since the R_i are linear arithmetic circuits) with $\vec{w}_p \in \mathbb{K}^n, \vec{s}_p \in \mathbb{K}^{\rho_i}$, or is a probe on the outer circuits and is of the form:

$$p = f_p(\vec{z}_1, \vec{z}_2) + \vec{r}_3^{\mathsf{T}} \cdot \vec{s}_p \tag{7}$$

where $\vec{z}_i := R_i(\vec{x}_i, \vec{r}_i)$ for $i \in \{1, 2\}$ with $\vec{s}_p \in \mathbb{K}^{\rho_3}$, and for some arithmetic function $f_p : (\mathbb{K}^n)^2 \to \mathbb{K}$. We show hereafter that we can still obtain an efficient and complete evaluation of SIS for those gadgets, which in turn yields an efficient verification of the considered security notions.

The verification technique for NLR-gadgets essentially consists in several iterations of the verification process for LR-gadgets used in Lemma 1. The steps of the technique are as follows. Suppose that we have a tuple of probes $\vec{P} = (p_1, \ldots, p_k, p_{k+1}, \ldots, p_d)$ where $(p_1, \ldots, p_k)$ are all of the form (7) while $(p_{k+1}, \ldots, p_d)$ are all of the form (6).

1. First, we apply the Gaussian elimination technique of Section 3.1 on the probes $(p_1, \ldots, p_k)$ with respect to the vector of randoms $\vec{r}_3$. This is possible since all of these probes respect the form (5) w.r.t. inputs $(\vec{z}_1, \vec{z}_2)$ and randomness $\vec{r}_3$. Specifically, let $S_3 := (\vec{s}_{p_1} \mid \vec{s}_{p_2} \mid \cdots \mid \vec{s}_{p_k})$, with $\vec{s}_{p_i}$ defined from (7), and let N_3 the permutation matrix such that $S'_3 = N_3 \cdot S_3$ is the row reduced form of S_3 (see Lemma 1). From this, we get a new derived tuple $\vec{P}' := N_3 \cdot \vec{P} = (p'_1, \ldots, p'_m, p'_{m+1}, \ldots, p'_k)$ and we know from Lemma 1 that each of the expression in $(p'_{m+1}, \ldots, p'_k)$ can be perfectly simulated by simply generating $k - m$ uniform random values. Thus, we end up with $(p'_1, \ldots, p'_m)$, which we need to perfectly simulate, and where each of the p'_i is of the form $f_{p'_i}(\vec{z}_1, \vec{z}_2)$ with no random values from $\vec{r}_3$, along with the remaining probes $(p_{k+1}, \ldots, p_d)$. We then construct the new tuple to simulate $\vec{P}'' = (p'_1, \ldots, p'_m, p_{k+1}, \ldots, p_d)$, which we rewrite as $\vec{P}'' = (p''_1, \ldots, p''_{m+d-k})$. Thus, in order to perfectly simulate the tuple of probes $\vec{P}$, we need to perfectly simulate the probes in $\vec{P}''$.
   We stress at this stage that each algebraic expression p''_i in $\vec{P}''$ is either of the form $p''_i = f_{p''_i}(\vec{z}_1, \vec{z}_2)$ with $f_{p''_i}$ a homogeneous bilinear form (this is of the first m coordinates resulting from Gaussian elimination) or of the form (6) (*i.e.* the probes on R_1 or R_2 that are not affected by the previous Gaussian elimination since they do not contain any randoms from $\vec{r}_3$).
2. For each p''_i in $\vec{P}''$ of the form $p''_i = f_{p''_i}(\vec{z}_1, \vec{z}_2)$, we factor its algebraic expression with respect to the vector of values $(\vec{x}_1 || \vec{r}_1)$. In other terms, we rewrite each p''_i as

$$p''_i = (\vec{x}_1 || \vec{r}_1)^{\mathsf{T}} \cdot \vec{h}_{p''_i} \tag{8}$$

where $\vec{h}_{p_i''}$ is a tuple of $n + \rho_1$ algebraic expressions of the form (6) w.r.t. $(\vec{x}_2, \vec{r}_2)$. We then construct a new tuple $\vec{P}_2 := (\vec{h}_{p_1''} \parallel \cdots \parallel \vec{h}_{p_m''})$ to which we append all the expressions p_i'' of the form (6) w.r.t. $(\vec{x}_2, \vec{r}_2)$ (*i.e.* probes from R_2).

3. We perform the same procedure as in the last step but this time factoring each p_i'' in $\vec{P}''$ of the form $p_i'' = f_{p_i''}(\vec{z_1}, \vec{z_2})$ with respect to $(\vec{x}_2||\vec{r}_2)$, rewriting each p_i'' as

$$p_i'' = (\vec{x}_2||\vec{r}_2)^\intercal \cdot \vec{g}_{p_i''} \tag{9}$$

From those expressions we define a new tuple $\vec{P}_1 := (\vec{g}_{p_1''} \parallel \cdots \parallel \vec{g}_{p_m''})$ where the coordinates of the $\vec{g}_{p_i''}$'s are of the form (6) w.r.t. $(\vec{x}_1, \vec{r}_1)$, to which we append all the expressions p_i'' of the form (6) w.r.t. $(\vec{x}_1, \vec{r}_1)$ (*i.e.* probes from R_1).

4. Recall from the first step that perfectly simulating $\vec{P}$ amounts to perfectly simulating $\vec{P}''$. We will prove later in this section that the input shares from $\vec{x}_1$ and $\vec{x}_2$ that are necessary and sufficient to produce a perfect simulation of $\vec{P}''$ are the same as the ones for a perfect simulation of $\vec{P}_1, \vec{P}_2$ constructed in the last two steps. Observe that all probes in $\vec{P}_1, \vec{P}_2$ respect the form (6), which is a special case of (5). Hence, we separately apply the Gaussian elimination technique of Lemma 1 on $\vec{P}_1$ with respect to $(\vec{x}_1, \vec{r}_1)$, and on $\vec{P}_2$ with respect to $(\vec{x}_2, \vec{r}_2)$. This provides us with the sets of input shares I_1 on $\vec{x}_1$ and I_2 on $\vec{x}_2$ that are respectively necessary and sufficient to produce a perfect simulation of the expressions in $\vec{P}_1$ and $\vec{P}_2$. These sets are therefore output as the necessary and sufficient sets of input shares for a perfect simulation of $\vec{P}$.

We state in the following lemma that the above verification method is complete (the proof is in appendix).

Lemma 2. *Let G be a 2-input n-share NLR-gadget. Let $\vec{P} = (p_1, \ldots, p_d)$ be a tuple of probes on G. Let $\vec{P}_1, \vec{P}_2$ be the tuples of linear expressions w.r.t. $(\vec{x}_1, \vec{r}_1)$ and $(\vec{x}_2, \vec{r}_2)$ obtained by applying the above method. The sets I_1, I_2 obtained by applying the method of Lemma 1 on $\vec{P}_1$ with respect to $(\vec{x}_1, \vec{r}_1)$ and separately on $\vec{P}_2$ with respect to $(\vec{x}_2, \vec{r}_2)$ are the sets of input shares necessary and sufficient to simulate $\vec{P}$.*

The verification method introduced above actually describes the procedure of the function SIS in the case of NLR-gadgets to determine the simulatability of a set of probes on such gadgets. We show in Section 4 how this technique is implemented in IronMask. We now present a concrete example of SIS execution on a set of probes on an NLR-gadget.

Example: Let us consider the following 2-share multiplication gadget (with inputs a and b, and output e) while taking $\mathbb{K} = \mathbb{F}_2$:

$$\begin{aligned} c_1 &= a_1 + r_a, & c_2 &= a_2 + r_a \\ d_1 &= b_1 + r_b, & d_2 &= b_2 + r_b \\ e_1 &= (c_1 * d_1 + r) + c_1 * d_2 \\ e_2 &= (c_2 * d_1 + r) + c_2 * d_2 \end{aligned}$$

The above gadget is an example of NLR-gadgets, and uses 3 random values: r_a is used to refresh the input sharing a, r_b is used to refresh the input sharing b, and r is used during the compression of the products into the output sharing e. The non-linear random values are r_a and r_b with respect

to e.
Suppose that we would like to verify one of the security properties defined in Section 2 using SIS. To do this, we need to be able to determine for each set of probes (formed of intermediate values and/or output shares) on the gadget, the exact set of input shares necessary and sufficient for a perfect simulation of all of the probes in the set. Let us consider for instance the following set of 2 probes on the gadget:

$$P = \{p_1 = c_1 * d_1 + r, \quad p_2 = c_2 * d_1 + r\}$$

We need to determine the set of input shares of a and b necessary to perfectly simulate probes in P. SIS will be executed in four steps as described earlier.
Step 1: get rid of the random values that are additive in the compression step (which are not additive to the shares of a and b), in this case it is the unique random value r . Using the Gaussian elimination technique, we construct a new set :

$$P' = \{p_1 + p_2 = c_1 * d_1 + c_2 * d_1, \quad p_2 = c_2 * d_1 + r\}$$

Since r only appears in p_2, this probe can be perfectly simulated by a uniform random value. Next we need to consider the simulation of the new set

$$P'' = \{c_1 * d_1 + c_2 * d_1\} = \{(a_1 + r_a) * (b_1 + r_b) + (a_2 + r_a) * (b_1 + r_b)\}$$

Step 2: factor the expressions in P'' with respect to the elementary variables of shares of a and random values which are additive to the shares of a and the constant term 1, in this case the variables $(a_1, a_2, r_a, 1)$. Since there is a single expression in P'', we can rewrite is as:

$$P'' = \{a_1 * (b_1 + r_b) + a_2 * (b_1 + r_b) + r_a * (b_1 + r_b + b_1 + r_b) + 1 * (0)\}$$

from which we construct the new set of the expressions multiplying $(a_1, a_2, r_a, 1)$

$$P_2 = \{b_1 + r_b, b_1 + r_b, 0, 0\}$$

Step 3: do the same thing with respect to $(b_1, b_2, r_b, 1)$:

$$P'' = \{b_1 * (a_1 + r_a + a_2 + r_a) + b_2 * (0) + r_b * (a_1 + r_a + a_2 + r_a) + 1 * (0)\}$$

from which we construct

$$P_1 = \{a_1 + a_2, 0, a_1 + a_2, 0\}$$

Step 4: determine the input shares of a necessary to simulate the expressions in P_1 and the shares of b necessary to simulate the expressions in P_2.

- for input a, we trivially need both input shares (a_1, a_2) to perfectly simulate expressions in P_1.
- for input b, we apply one step of Gaussian elimination with respect to r_b, to obtain the new set $P_2' = \{b_1 + r_b, 0, 0, 0\}$. We can see that we can perfectly simulate the single non-zero expression with a uniform random value. Thus, in this case, no shares of b are necessary to perfectly simulate P_2' and hence also P_2.

Hence, to perfectly simulate P_1 and P_2, we need both input shares of a and no shares of b. Thanks to Lemma 2, we can conclude that P can be perfectly simulated using both shares of a and no shares of b.

4 Efficient Verification

In this section, we introduce IronMask, a new tool that we developed to check probing and random probing security properties using the algorithms presented in Section 3. The implementation of IronMask considers currently a finite field $\mathbb{K}$ of characteristic 2, it can be easily extended in the future to any finite field since the verification methods introduced in the previous sections work in any finite field $\mathbb{K}$. IronMask is written in C, and the only external libraries it depends on are the GNU Multiple Precision Arithmetic Library (GMP) and the POSIX Threads (pthreads) library.

4.1 Data Representation

```
#shares 2
#in a b
#randoms r0
#out c

m0 = a0 * b1
t0 = r0 + m0
m1 = a1 * b0
t1 = t0 + m1

m2 = a0 * b0
c0 = m2 + r0

m3 = a1 * b1
c1 = m3 + t1
```

(a) 2-share ISW multiplication

```
#shares 3
#in a
#randoms r0 r1 r2
#out d

d0 = a0 + r0
d0 = d0 + r1

d1 = a1 + r0
d1 = d1 + r2

d2 = a2 + r1
d2 = d2 + r2
```

(b) 3-share refresh

Fig. 1: Masking gadgets written in IronMask's syntax

IronMask takes as input gadgets written in a simple syntax to describe circuits, borrowed from VRAPS [13]: a gadget is a list of assignments of additions or multiplications into variables, alongside directives to specify the number of shares, the inputs, the outputs and the randoms. Figure 1 illustrates our input syntax on a 2-shares ISW multiplication (Figure 1a) and a 3-share refresh gadget (Figure 1b). In Figure 1a, the variables `a0`/`b0` (resp. `c0`) and `a1`/`b1` (resp. `c1`) are the 1st and 2nd shares of the input `a`/`b` (resp. output `d`). Similarly to maskVerif, the syntax `![ expr ]` can be used to stop the propagation of glitches in the robust probing model. For instance, `tmp = a0*b0` could be replaced by `tmp = ![ a0*b0 ]`, in which case `tmp` would leak `a0*b0` instead of leaking `a0` and `b0` separately.

Internally, IronMask represents each wire of the gadget as an array of integers composed of three parts. The first ℓ parts correspond to linear dependencies on the inputs of the gadget: if the k^{th} bit of the n^{th} element is set to 1, then the wire depends linearly on the k^{th} share of the n^{th} input. The second part is a bitvector, where the k^{th} bit set to 1 indicates a linear dependency on the k^{th} random of the gadget. Finally, the third part is a bitvector as well, where the k^{th} bit set to 1 indicates a linear dependency on the k^{th} quadratic monomial appearing in the symbolic expressions

of the gadget wires. For instance, the internal representation of the wires `a0`, `a1`, `r0`, `m3`, `t1` and `c1` of Figure 1a are as follows:

```
     inputs       randoms       mults
a0: [ 1, 0,          0,        0,0,0,0 ]
a1: [ 2, 0,          0,        0,0,0,0 ]
r0: [ 0, 0,          1,        0,0,0,0 ]
m3: [ 0, 0,          0,        0,0,0,1 ]
t1: [ 0, 0,          1,        1,1,0,0 ]
c0: [ 0, 0,          1,        0,0,1,0 ]
```

With an additional data structure storing the operands of each multiplication:

```
0: a0 * b1            1: a1 * b0
2: a0 * b0            3: a1 * b1
```

Using this internal representation enables efficient operations down the line: the linear dependencies of a wire on the input shares are accessible with a single operation, the number of such input shares is efficiently obtained by counting the number of bits to one in its first element (or first two elements for 2-input gadgets), and `xor`ing two wires, which is one of the basic operations of our Gaussian elimination, can be easily done by `xor`ing pointwise the arrays representing them.

To model glitches in the robust probing model, we use the same glitch model as in [3]. Namely, we consider that an expression $a + b$ (resp $a * b$) leaks a and b separately, instead of leaking $a + b$ (resp $a*b$). Registers (usually called *flip-flops*) can be used to stop the propagation of these glitches. In IronMask, when taking glitches and transitions into consideration, each wire is represented by an array of arrays instead of a single array, since the leakage of an assignment is the union of the leakages of its right-hand side operands. For instance, the wire `c0` in Figure 1a in the presence of glitches is represented as:

```
       inputs       randoms       mults
c0: [ [ 0, 0,          1,        0,0,0,0 ],
      [ 1, 0,          0,        0,0,0,0 ],
      [ 0, 1,          0,        0,0,0,0 ] ]
```

If a flip-flop was added to `m2` to stop the propagation of glitches by doing `m2 = ![a0*b0]`, then the robust leakage of `c0` would become:

```
       inputs       randoms       mults
c0: [ [ 0, 0,          1,        0,0,0,0 ],
      [ 0, 0,          0,        0,0,1,0 ] ]
```

4.2 Basic Verification

In this section, we present the procedures implemented in IronMask for the verification of probing and random probing properties. Recall that in Section 2, we give definitions of all the security properties based on a single building block SIS: a primitive that, given a set of probes (internal probes and output probes), determines the input shares necessary for a perfect simulation of these probes. Thus, to verify any security property, IronMask uses a concrete implementation of the function SIS based on the algebraic characterization techniques discussed in Section 3.

Gadgets with linear randomness. For the verification of LR-gadgets introduced in Section 3.1 (*i.e.* gadgets in which all random values are additive), IronMask relies on the SIS_LR procedure (Algorithm 1). This procedure is a direct application of the result presented in Lemma 1. We recall that Algorithm 1 in IronMask considers currently a finite field of characteristic 2.

Algorithm 1 SIS_LR returns the input shares that are leaked by the tuple $\vec{P}$ with expressions of the form (5), assuming ℓ input sharings

```
1: procedure GaussElimination(P⃗)
2:     for each probe p_i of P⃗ do
3:         if p_i contains at least one random variable then
4:             r ← choose (any) one random variable in p_i
5:             for each probe p_j of P⃗ with i ≠ j do
6:                 if p_j contains r then
7:                     p_j ← p_i + p_j
8:             p_i ← r
9: procedure shares(P⃗)
10:    I_1 ← ∅, ..., I_ℓ ← ∅
11:    for each probe p_i of P⃗ do
12:        Add all input shares in p_i of each input j to I_j
13:    return I_1, ..., I_ℓ
14: procedure SIS_LR(P⃗)
15:    P⃗' ← GaussElimination(P⃗)
16:    return shares(P⃗')
```

As in Lemma 1, a Gaussian elimination is first performed on the tuple by the procedure GaussElimination, after which each probe of the input tuple is either "replaced" by a random r (as shown on line 8 of the procedure GaussElimination), or contains one or more input shares and no random values. In fact, what we mean by replacing the probe p_i by a random value r on line 8 is that after eliminating r from the expressions of all other expressions p_j in the same tuple (loop from line 5 to 7 where the instruction $p_j \leftarrow p_i + p_j$ aims to remove r from p_j in a finite field of characteristic 2), we end up with r only appearing in the expression of p_i and so as explained in the proof of Lemma 1, simulating p_i amounts to generating r uniformly at random without the need for any other variables. We represent this by replacing the expression of p_i by the single random value r. Then, the shares leaked by the input tuple can be found on the probes that do not contain any randoms using the procedure shares. The latter actually corresponds to an implementation of the function shares(.) used in Lemma 1.

Gadgets with non-linear randomness. For NLR-gadgets (*i.e.* gadgets performing non-linear operations on input shares mixed with randomness), IronMask uses the SIS_NLR procedure (Algorithm 2), which implements the four steps described in Section 3.2. As mentioned in Section 3.2, SIS_NLR currently only supports gadgets with two input sharings, but can be extended in the future to ℓ input sharings.

First, SIS_NLR performs Gaussian elimination with respect to the vector of output randoms (*i.e.* $\vec{r_3}$), using a modified version of GaussElimination that takes as inputs the randoms to use for the elimination. This corresponds to step 1 of Section 3.2. The modified probes in $\vec{P'}$ do not contain any more random values of $\vec{r_3}$. Next, two new tuple of probes $\vec{P_1}, \vec{P_2}$ are constructed

Algorithm 2 SIS_NLR returns the input shares that are leaked by the tuple P in an NLR-gadget refreshing its output with the randoms $\vec{r_3}$ (*c.f.* sec. 3.2), assuming 2 input sharings

procedure SIS_NLR($\vec{P}, \vec{r_3}$)
  $\vec{P'} \leftarrow$ GaussElimination($\vec{P}$, $\vec{r_3}$)
  $\vec{P_1} \leftarrow (\,), \vec{P_2} \leftarrow (\,)$
  for each probe p_i in $\vec{P'}$ **do**
    if p_i contains no randoms of $\vec{r_3}$ **then**
      $(\vec{P'_1}, \vec{P'_2}) \leftarrow$ FactAndExtract(p_i)
      $(\vec{P_1}, \vec{P_2}) \leftarrow (\vec{P_1}||\vec{P'_1}, \vec{P_2}||\vec{P'_2})$
  $I_1 \leftarrow$ SIS_LR($\vec{P_1}$), $I_2 \leftarrow$ SIS_LR($\vec{P_2}$)
  return I_1, I_2

from the probes in $\vec{P'}$, using the FactAndExtract procedure, which corresponds to the factoring technique discussed in steps $2-3$ of Section 3.2. The pseudo-code of this function is left out for conciseness. We thus get two tuples P_1 and P_2 containing input shares, randoms and refreshed input shares from each input. Since those variables are linear, we can use the initial SIS_LR procedure to extract the input shares that they leak.

Verification of security properties. Checking any probing or random probing property (*e.g.* NI, SNI, RPC, RPE, ...) consists in enumerating tuples of probes, using SIS_LR or SIS_NLR to get the input shares that they leak (we abbreviate with SIS and suppose that we make a call to the correct algorithm for LR-gadgets and NLR-gadgets), and take some action in consequence (see Section 2.2). In the following, we shall call a *t-failure tuple* (or simply a *failure tuple* when t is not made explicit) any tuple of probes that leaks more than t input shares of one or more input sharings (*i.e.* for which SIS outputs a set or more of cardinality strictly greater than t).

For instance, to verify if an n-share gadget is t-NI, we enumerate all tuples of size t, and make sure that none of them is a t-failure tuple (Algorithm 3). This corresponds to the first row of Table 1. Or, to verify the (p, ε)-RPS* of an n-share gadget G in the random probing model (first row of Table 2), we need to compute the coefficients c_i from equation (3) of the failure probability function $f(p) = \varepsilon$ as explained in Section 2.2. This corresponds to enumerating all the tuples of probes (excluding the output wires) of size 1 to a threshold $c_{\max}$ and count how many leak more than t-shares (after $c_{\max}$, upper and lower bounds on $f(p)$ are obtained, we refer the readers to [13] for more details on the process and the threshold). This corresponds to the procedure depicted in Algorithm 4.

Algorithm 3 Is_t_NI returns true if G is t-NI and false otherwise, assuming G has ℓ input sharings

procedure Is_t_NI(G, t)
  for each tuple $\vec{P}$ of size t in G **do**
    $I_1, \ldots, I_\ell \leftarrow$ SIS($\vec{P}$)
    if $|I_1| > t$ **or** $\ldots$ **or** $|I_\ell| > t$ **then**
      return false
  return true

Enumerating all tuples becomes impractical as soon as gadgets start growing larger than a few hundred variables, since the number of tuples of size k in a gadget containing s variables is $\binom{s}{k}$.

Algorithm 4 GetCoeffsRPS* returns an array of c_{max} cells where the k^{th} index contains the number of failure tuples of k probes on n-share gadget G with ℓ input sharings

```
procedure GetCoeffsRPS*(G, c_max)
    coeffs ← [0, ..., 0] of size c_max
    for k = 1 to c_max do
        for each tuple P⃗ of k probes on G do
            I_1, ..., I_ℓ ← SIS(P⃗)
            if |I_1| > n − 1 or ... or |I_ℓ| > n − 1 then
                coeffs[k] += 1
    return coeffs
```

For instance, checking that a 9-share masked ISW multiplication containing 279 variables is 8-NI requires enumerating $\binom{279}{8} \approx 8 \times 10^{14}$ tuples, which is not far from being out of reach for modern computers.

The rest of the section is organized as follows. In Section 4.3, we address dimension reduction techniques proposed in [11,19] to reduce the search space of the enumerated tuples. In Section 4.4, we present some optimizations of our implementations that make verification faster by reducing the cost of SIS_LR (since the latter is also a building block for SIS_NLR) and parallelizing our procedures. Finally, in Section 4.5, we introduce a constructive algorithm to generate failures without having to enumerate all tuples in the case of linear gadgets.

4.3 Dimension Reduction

Checking any probing or random probing property requires enumerating many tuples. For instance, for a gadget G made of s variables, $\binom{s}{t}$ tuples need to be checked to assess whether G is t-NI or not. To reduce the number of tuples that have to be considered, we remove some variables from the search. First, as proposed in [11] and further explained in [19], elementary deterministic probes can be removed when checking any probing or random probing property. Then, when checking for probing properties only, we use the "reduced sets" optimization proposed in [19], which consists in eliminating some "less powerful" variables from the search. In appendix, we recall the principle of those two optimizations, and show how to make the first one work in the random probing model, and why the second one cannot be used in this model. Note that the dimension reduction technique is proved to be sound in [19], which means that our verification technique implementing the optimization remains sound.

4.4 Implementation Optimizations

On-the-fly Gaussian Elimination. In order to find all failures of a given size, we enumerate all the tuples of that size, and apply the SIS procedure on each of them. This means that a full Gaussian elimination has to be performed on each tuple. However, we generate the tuples in lexicographic order, which mean that two consecutive tuples only differ only by their last elements, and, in most cases, only by their very last element. For two consecutive tuples, it is thus very likely that most of the Gaussian elimination will be identical. We take advantage of this by implementing our Gaussian elimination on the fly: for each tuple, we only recompute the elimination on the elements that differ from the previous tuple.

Table 3: RPS* performance of our new constructive algorithm against our traditional enumerative one

Gadget	Shares	#wires	Enumerative		Constructive	
			c_{max}	Verif time	c_{max}	Verif time
ISW refresh	8	140	8	5min	8	3sec
					10	6min
ISW add	7	224	7	18min	8	2sec
					9	2min
$n \log n$ refresh	8	100	9	1min	9	2sec
					11	2min
ISW mult refreshed	6	297	6	2min	6	12min
			7	38min		

The cost of the Gaussian elimination for a single tuple of k elements of a gadget containing s inputs and randoms is $\mathcal{O}(sk^2)$. Performing the elimination on-the-fly brings the amortized complexity down to $\mathcal{O}(sk)$.

A similar, slightly more efficient technique has been used by [19] to speed up their implementation. They used a revolving-door algorithm to generate the tuples, so that each consecutive tuple differs by exactly one element, which allowed them to amortize the cost of their analysis. However, we cannot use this revolving-door algorithm because when changing the i^{th} element of a tuple, the Gaussian elimination needs to be recomputed from this i^{th} element up to the end of the tuple.

Parallelization. Recall that we generate the tuples in lexicographic order, which admits an efficient *unranking* algorithm. This means that we can easily compute what the j^{th} tuple of size k is, for any j and any k. Multi-threading the verification of n tuples is thus trivial: to run l threads in parallel, the j^{th} thread starts with the $\lfloor j \times n/l^{th} \rfloor$ tuple, and verifies the next $\lfloor n/l \rfloor$ tuples.

Our implementation is multi-threaded in this fashion using POSIX threads, provided by the `pthread` library. In order to be transparent from the properties' point of view (*e.g.*, from Algorithms 3 and 4), the multi-threading is done inside SIS. To this end, we use a few mutexes, which incur an overhead in the random probing model: the more failures a gadget contains, the less of a speedup multi-threading offers. Although it would not be hard to implement multi-threading on the properties' side rather than in SIS, we opted for readability and maintainability of the code, at the expense of a bit of performance.

4.5 Constructive Approach

The enumerative approach of Section 4.2 generates a lot of tuples that are trivial non-failures because they do not contain enough shares to be failures, or their shares are masked by random variables. To overcome this issue, we designed a constructive algorithm to only generate potential failures. We give a detailed description of this algorithm in Appendix D.

This constructive algorithm is faster than the traditional enumerative algorithm of Section 4.2 when checking $(n-1)$-NI and RPS* properties for linear gadgets. Table 3 shows the exact performance improvements when checking the RPS* property on some common linear gadgets (ISW refresh [32], $n \log n$ refresh [9], and an "ISW addition" made of a share-wise addition preceded by an ISW refresh of each input), and on an ISW multiplication with a circular refresh on one of the

inputs. On linear gadgets, the constructive algorithm can go about 2 coefficients further than the enumerative one within the same time, thus producing much more precise results.

Furthermore, the constructive algorithm enables the verification of larger, previously out of reach, gadgets. For instance, a 9-share ISW addition contains 243 variables, and thus contains $\binom{243}{9} \approx 7 * 10^{15}$ tuples of size 9 (for 8-RPS*), which is clearly beyond the capabilities of the enumerative algorithm. Yet, our constructive algorithm is able to generate all of its failures of size 9 in 7 minutes.

However, on multiplication gadgets, the constructive algorithm is slower than the enumerative one. Table 3 illustrates this on an ISW multiplication with an ISW refresh on one of the inputs. Additionally, the constructive algorithm does not perform well in terms of performance when checking t-NI or t-RPS* with $t < n - 1$, as well as SNI, RPC and RPE. We explain in Appendix D why this is the case.

5 Evaluation

To showcase IronMask, we start in Section 5.1 by providing new bounds for the maximum RPE leakage probability tolerated by some common gadgets (in the random probing model). Then, we compare the scope (Section 5.2) and performance (Section 5.2) of IronMask and existing verification tools: VRAPS and STRAPS in the random probing model, and maskVerif, matverif and SILVER in the probing model. The description files of the gadgets tested in the following sections are publicly available on IronMask's GitHub repository.

5.1 New Random Probing Expandability Results

So far, VRAPS [13] was the only tool verifying the (t, p, ε)-RPE property. IronMask is several orders of magnitude faster than VRAPS, in addition to being complete (IronMask avoids failure false positives *i.e.* detected failure tuples which are not really failures, unlike VRAPS), allowing us to compute more precise bounds for the coefficient of the failure function $f(p) = \varepsilon$ (*c.f.* Section 2.2) and hence more precise bounds on the tolerated leakage probability. In particular, we consider the ISW multiplication and refresh [32], the $n \log n$ refresh [9], the circular refresh [7], as well as the addition, copy, and multiplication from [16, Section 6.2]. Additionally, we consider addition (resp. copy) gadgets obtained by doing an ISW or $n \log n$ refresh on one of the inputs followed by a simple addition (resp. copy). Finally, we also evaluate a double-SNI multiplication [31] made of an ISW multiplication where one of the inputs is refreshed using a n circular refreshes [25] (with n shares).

The result are shown in Table 4. For the t parameters, we used $t = \lfloor (n-1)/2 \rfloor$ (with n shares), as recommended by [16]. For large gadgets, we cannot compute precisely the maximum leakage probability tolerated in reasonable time. Instead, like [13], we compute all failures up to a given size c_{max}, which allows us to obtain upper and lower bounds for the leakage probability.

For ISW multiplication, our results improve previous results from [13] (obtained with VRAPS) in two ways: by increasing the value $c_{\max}$ of the verification we obtain tighter bounds on the failure event function $f(p)$ and thus tighter intervals (and even exact values in some cases) for the tolerated leakage probability. Plus, thanks to the completeness of the verification of IronMask by avoiding failure false positives (unlike VRAPS), we obtain better values for the estimated tolerated leakage probability (by better we mean higher probability values). For example, our results show that the (exact) tolerated leakage probability of the 6-share ISW multiplication is 2^{-12} instead of the 2^{-13} lower bound of [13]. We obtain similar improvements for the 5-share gadgets of [16] which are today

Table 4: Maximum t-RPE leakage probabilities tolerated by some common masking gadgets

Gadget		Shares	t	Ampl. order	c_{max}	#wires	$\log_2$ maximum tolerated proba.	Verif. time
Linear Randomness								
ISW [32]	mult	5	2	3/2	6	180	-10.54	24min
		6	2	3/2	5	267	-12.00	13min
		7	3	2	5	371	[-10.45, -8.73]	28min
	refresh	5	2	3	10	50	-4.28	2min
		6	2	3	8	75	[-4.81, -4.61]	5min
		7	3	4	7	105	[-5.50,-4.01]	21min
	add	5	2	3	7	110	[-6.48,-4.70]	11min
		6	2	3	6	162	[-7.81,-5.03]	17min
		7	3	4	6	224	[-8.47,-4.15]	3h
	copy	5	2	3	6	105	[-5.92,-5.54]	12min
		6	2	3	5	156	[-6.92,-5.93]	24min
		7	3	4	4	217	[-8.02,-3.87]	33min
$n \log n$ [9]	refresh	4	1	2	30	30	-5.27	1sec
		8	3	4	7	100	[-5.42,-4.36]	18min
	add	4	1	2	8	68	-5.40	4min
		8	3	4	6	216	[-8.40,-4.40]	4h
	copy	4	1	2	6	64	-6.96	27sec
		8	3	4	4	208	[-7.94,-4.25]	55min
circular refresh [7]		5	2	3	25	25	-4.84	1sec
		10	4	3	8	50	-5.21	1min
[16]	add	5	2	3	9	55	[4.67,-4.42]	10min
	copy	5	2	3	6	60	-6.17	41sec
Non-linear Randomness								
Double-SNI ISW		4	1	2	5	190	-9.85	5min
		5	2	5/2	5	305	[-10.01,-8.09]	31min
[16] mult		5	2	3	6	405	[-9.67,-7.66]	31h

the gadgets giving the best asymptotic complexity of $\mathcal{O}(\kappa^{3.23})$ for the expansion strategy with a constant leakage probability and for a target random probing security of $2^{-\kappa}$. Our results improve the lower bound on the tolerated leakage probability for those gadgets from 2^{-12} to $2^{-9.67}$. For all the other gadgets in the table, we report the first verification results of the RPE property.

5.2 Comparison with State-Of-The-Art Tools

We compare IronMask to six carefully chosen state-of-the-art tools: maskVerif [5,6,3] (and its extension scVerif [8]), matverif [19], SILVER [33], VRAPS [13], and STRAPS [20], with which our new tool IronMask shares the following features:

- does not rely on any gadget's structure (unlike *e.g.*, `maskComp` [6], `tightPROVE` [15], `Tornado` [14]),
- verifies probing or random probing-like security notions.

We discuss the properties that are concretely verified, and provide some benchmarks to highlight the main differences with IronMask.

Scope. Introduced in 2015 [5] and then extended multiple times ([6,3]), maskVerif is the very first tool able to verify reasonable higher-order masking schemes. Based on a symbolic representation of

leakage, it integrates the language-based verification of (robust) probing security and (S)NI notions with or without leakage on registers transitions. One step further, the latest extension of maskVerif, referred to as scVerif [20], captures even more hardware side effects, potentially configurable by the user [8]. Compared to our proposal, maskVerif includes tricks to verify bigger circuits (*e.g.*, s-boxes, block encryption scheme) but fails to provide a complete verification as soon as the randomness is not linear (*i.e.*, failure false positives may be produced).

In the same vein, matverif [19] targets the same properties as maskVerif. It features a new method to obtain a complete verification (*i.e.*, without any failure false positive) for specific circuits (*e.g.*, ISW multiplications) and significantly improve its performance thanks to dimension-reduction strategies. In terms of supported gadgets, matverif is more limited than maskVerif and IronMask, as it does not support gadgets with non-linear randomness at all. Unlike our proposal and similarly to maskVerif, matverif focuses only on the verification of probing-like properties.

Following a different strategy, SILVER [33] was built to verify the physical security of hardware designs. It takes as input either a Verilog implementation or an instruction list and checks the probing, (S/PI)NI notions in the standard and robust models, as well as the uniformity of some output sharing. On the one hand, it outperforms the capacities of maskVerif by offering a complete verification based on a symbolic and exhaustive analysis of probability distributions and statistical independence of joint distributions. On the other hand, its verification is significantly slower than that of maskVerif.

Introduced in 2020, VRAPS is the first tool to verify random-probing-like properties [13] (to the best of our knowledge). Written in Python and SageMath, it was built to evaluate the RPE security of some base gadgets, in order to assess the global security of the expanding compiler of [13]. Specifically, VRAPS detects all the leaking tuples within an implementation with respect to the RPS*, RPE1, RPE2 and RPC security properties introduced in Section 2. Nevertheless, it suffers from low performance and, unlike IronMask, can generate failure false positives for both gadgets with linear and non-linear randomness. VRAPS supports more gadgets than IronMask which is limited to LR-gadgets and NLR-gadgets. Nevertheless, to the best of our knowledge, all the masking gadgets in the literature fit the latter representations. While VRAPS can additionally (directly) verify bigger gadgets (*i.e.*, composition of atomic gadgets), in practice, the performance and the completeness would be very low. In addition, the verification of atomic gadgets using IronMask already makes it possible to obtain secure global circuits since once individually verified (for probing or random probing properties), they can be safely composed [6,**?**].

Finally, STRAPS is a very recent tool designed to verify random probing-like properties [20]. In particular, it was built to compute the distribution of a gadget's input sets of shares with respect to the output observations and the leakage probability of each internal wire. In its deterministic mode, it relies on maskVerif as a basic primitive. One step further, it integrates a probabilistic mode, based on Monte-Carlo methods, which significantly improves the performance by avoiding a full exploration and limiting the analysis to selected tuples. While the probabilistic mode can allow increased performance and thus more accurate results for random probing properties, it uses a set of rules from maskVerif as a building block . These rules by construction do not provide complete verification, which implies that the verification method of STRAPS is not complete either.

Table 5 recalls the categories of properties (as in Section 2) that are verified by the aforementioned tools on higher-order masked implementations. It additionally specifies the consideration of hardware effects, *i.e.* glitches (captured in the robust probing model). A green check (✓) means that the row tool verifies the column property. On the contrary, a red cross (✗) means that the

Table 5: Verified security properties on higher-order masked implementations for carefully chosen state-of-art automatic tools.

Tools	probing-like		RP-like	
	soft	robust	soft	robust
maskVerif	✓	✓	✗	✗
scVerif	✓	✓	✗	✗
matverif	✓	✓	✗	✗
SILVER	✓	✓	✗	✗
VRAPS	✓	✗	✓	✗
STRAPS	✗	✗	✓	✗
IronMask	✓	✓	✓	✓

column property is not handled by the row tool. We can see that IronMask is the first tool to provide verification for probing-like properties and also for random probing-like properties in the standard model and in presence of glitches (robust model).

Additionally IronMask offers a complete verification method for gadgets with linear randomness as well as for most deployed gadgets with non-linear randomness (and in particular all known multiplication gadgets). The only other tool providing complete verification for such gadgets is SILVER but this is achieved by an exhaustive approach making its running time quickly prohibitive (see comparison hereafter).

Performance. We evaluate the performance of IronMask compared to other state-of-the-art verification tools in both the probing and random probing models.

Probing Model. We compared the time required by IronMask, maskVerif and matverif to check that some commonly-used gadgets are $(n-1)$-NI and $(n-1)$-SNI (abstracted NI and SNI hereafter for conciseness). In particular, the gadgets we considered are: the ISW multiplication [32], the double-SNI multiplication [31, 21] using an ISW multiplication and a circular refresh [25] on one of the inputs, the new NI and SNI multiplications from [19], and the $n \log n$ refresh [9]. The results are presented in Table 6.

We used the multi-threaded version of each tool, setting the maximal number of cores to use to 4, to give a fair chance to maskVerif: while IronMask and matverif can use an arbitrary number of cores, maskVerif is limited to 4 cores. We evaluated several masking orders for each gadget in order to highlight the scaling of each tool. To save time, we did not run the verification of gadgets when it would have taken more than a few hours. Finally, to analyze a gadget, matverif needs probes description files to be generated using a SageMath script, and the main program to be recompiled. While the incurred additional time was ignored in [19], we take it into account so that time we report reflects the actual time that a user would need to check those gadgets.

Before discussing the results, we recall that the three tools are not functionally equivalent. maskVerif can handle any circuit, while IronMask is limited to LR-gadgets and NLR-gadgets as characterized in Section 3, and matverif only handles LR-gadgets. On the other hand, the verification of IronMask is complete for both types of gadgets, while maskVerif's is not. In addition, while IronMask is limited to LR-gadgets and NLR-gadgets, to the best of our knowledge, all the masking gadgets in the literature fit the latter representations, and the verification of atomic gadgets already makes it possible to obtain secure global circuits through composition [6, **?**].

Table 6: Comparison of the performance of IronMask, maskVerif and matverif on higher-order masked gadgets. The multithreaded versions of each tools were used, with the maximum number of threads set to 4. `N/A` means that a tool cannot check a gadget, whereas - means that a tool was not evaluated on a gadget because we deemed it too slow.

Gadget	**Type**	**Shares**	**Property**	**Verification time**		
				IronMask	maskVerif	matverif
ISW mult	LR	7	NI	7sec	1min30sec	24sec
			SNI	8sec	3min56sec	25sec
		8	NI	4min 6sec	2h 10min	5min 19sec
			SNI	5min 15sec	6h 30min	5min 15sec
		9	NI	2h 22min	-	2h 3min
			SNI	3h 7min	-	1h 58min
ISW mult refreshed	NLR	6	NI	2sec	3sec	N/A
			SNI	3sec	10sec	N/A
		7	NI	3min 41sec	1min 50sec	N/A
			SNI	6min	8min 16sec	N/A
		8	NI	8h 52min	2h 2min	N/A
			SNI	14h 46min	10h 4min	N/A
NI mult [19]	LR	7	NI	1sec	1min50sec	5sec
		8		5sec	2h 10min	9sec
		9		2min50sec	-	40sec
		10		6h 28min	-	1h 40min
SNI mult [19]	LR	7	SNI	1sec	6min	8sec
		8		46sec	6h 26min	17sec
		9		24min	-	4min 37sec
		10		24h	-	1h 54min
refresh $n \log n$	LR	9	NI	1sec	$<$ 1sec	1sec
			SNI	24sec	2sec	1sec
		10	NI	1sec	$<$1sec	10sec
			SNI	16min	9sec	10sec
		11	NI	1sec	$<$1sec	3min
			SNI	7h 50min	1min	3min
		12	NI	1sec	$<$1sec	3h 35min
			SNI	-	5min	1h 52min

Overall, the three tools have similar performance, and allow to analyse gadgets up to similar masking orders, although each tool has its strengths and weaknesses. For instance, maskVerif performs better on LR-gadgets, while matverif shines on LR multiplications. In the following, we investigate in more details the relative speed of each tool.

To analyse the performance of IronMask compared to matverif, five main factors need to be taken into account. First, matverif's method to verify a tuple of probes on an LR-gadget has a complexity linear in the tuple's length, while our SIS_LR method has the complexity of a Gaussian elimination which in our implementation is quadratic in the tuple's length. Second, the dimension reduction performed by IronMask is faster than that of matverif in most cases, probably because ours is written in C and matverif's in SageMath. Third, our dimension reduction often removes more wires, resulting in fewer tuples to consider. Fourth, matverif needs to be recompiled for each gadget. Fifth and last, on linear gadgets (add and copy) and when checking $(n-1)$-NI, IronMask uses the constructive algorithm (see Section 4.5), which is much faster than any enumerative algorithm.

It can be observed through Table 6 that the scale of the speedups offered by IronMask compared to matverif and maskVerif depends mostly on the structure and nature of the gadgets. We will explain hereafter the reason for this scaling depending on the tested gadgets.

matverif tends to be slower than IronMask at smaller masking orders, and at any order when checking $(n-1)$-NI on linear gadgets. However, at the highest masking orders, the cost of the dimension reduction and the recompilation of matverif become negligible compared to the main enumerative algorithm. matverif thus becomes faster than IronMask, thanks to its complexity linear in the tuples' length (compared to the quadratic complexity of IronMask's SIS_LR primitive).

On the standard ISW multiplication and the multiplication from [19], both matverif and IronMask outperform maskVerif thanks to the dimension reduction they use (see Section 4.3), which is not implemented in maskVerif. For instance, the 7-order ISW multiplication contains initially 220 variables, but only 77 remain after the dimension reduction.

On the double-SNI ISW multiplication with a circular refresh on the inputs, the dimension reduction is much less potent. For the 6^{th}-order double-SNI multiplication, 175 probes must thus be taken into account, against 57 for the standard 6^{th}-order ISW multiplication. As a result, maskVerif and IronMask have similar performance on this gadget.

The $n \log n$ refresh is obviously NI at any order, since at no point are multiple secret shares part of the same probe. maskVerif seems to have a special rule to detect that, resulting in a verification of NI that is almost instantaneous. We did not add this special rule in IronMask, but our constructive algorithm (presented in Section 4.5) is able to very quickly detect that the gadget is indeed NI. To check that this gadget is SNI, however, both matverif and IronMask enumerate all tuples, which becomes very expensive as the masking order grows. As a result, checking that the 11^{th}-order gadget is SNI with IronMask would require at least a few days. On the other hand, maskVerif does not need to enumerate all tuples, and is able to much quickly determine that this gadget is SNI, taking just 5 minutes to do so at order 11.

Table 7: Verification time of NI and SNI verification of the ISW multiplication and $n \log n$ refresh by IronMask and SILVER

Gadget	**Shares**	**Property**	**Verification time**	
			IronMask	SILVER
ISW mult	4	NI	<1sec	1sec
		SNI	<1sec	2sec
	5	NI	<1sec	9min
		SNI	<1sec	14min
	6	NI	<1sec	>10h
		SNI	<1sec	>10h
refresh $n \log n$	6	NI	<1sec	8sec
		SNI	<1sec	20sec
	7	NI	<1sec	7min
		SNI	<1sec	14min
	8	NI	<1sec	>10h
		SNI	2sec	>10h

As mentioned earlier, SILVER, while being the only tool mentioned here that is complete on any gadget, suffers from severe performance limitations. This is illustrated in Table 7, which shows

that SILVER is several orders of magnitude slower than IronMask on the ISW multiplication and the $n \log n$ refresh.

Random Probing Model. Table 8 shows the time needed by VRAPS and IronMask to compute the maximum tolerated leakage probability of the ISW multiplications when setting c_{max} to 4 (which is the maximum that is computable by VRAPS in reasonable time). IronMask was not multi-threaded in this benchmark, and we recall that VRAPS does not support multi-threading. IronMask is several orders of magnitude faster than VRAPS in addition to being more precise (since VRAPS can incorrectly classify tuples as failures). The performance gains are explained by several factors. First, IronMask is written in C, whereas VRAPS is written in SageMath. Second, IronMask uses a complete technique based on Gaussian elimination to determine if tuples are failures, whereas VRAPS uses SageMath's symbolic calculus to iteratively apply simplification rules inspired from maskVerif. Third, IronMask allocates less memory, and performs its Gaussian elimination on the fly (see Section 4.4), whereas VRAPS allocates chunks of memory to store batches of tuples, and restarts the simplifications from scratch for each tuple.

Table 8: Performance of t-RPE verification of IronMask and VRAPS on ISW multiplication gadgets at orders 4 to 6

Shares	**t**	$\mathbf{c_{max}}$	**#wires**	$\log_2$ **maximum tolerated proba**	**Verification time**	
					IronMask	VRAPS
5	2	4	180	[-11.00,-10.67]	3sec	1h 15min
6	2	4	267	-13	17sec	24h
7	3	4	371	[-12.00,-7.83]	24sec	24h

The performance gains of IronMask over VRAPS have two main benefits. First, IronMask can be more useful for prototyping, since for small c_{max} it can provide approximate results within a few seconds. Second, IronMask can compute exact and more precise results by increasing c_{max}, as shown in Table 4.

6 Related Work

Many tools have been implemented in the past few years to verify software and hardware masked implementations. While we do not intend to provide an exhaustive list, we briefly recall the main lines of works for the verification of probing-like and random-probing like properties.

In 2012, Moss *et al.* [34] design the first automatic type-based masking compiler to provide first-order security against DPA. Following this seminal work, Bayrak *et al.* [10] investigate SMT-based method to evaluate the statistical independence between leakage and secrets. Eldib, Wang, and Schaumont [27] extend it to the verification of higher-order targets using a notion very similar to non-interference. Nevertheless, the complexity of their model counting approach restricts it to small masking orders.

Barthe *et al.* [5] then formalize the connection between the security of masked implementation and probabilistic non-interference. Their method makes it possible to overcome the combinatorial explosion of observation sets for high orders. The resulting tool, maskVerif, thus verifies reasonable circuits at reasonable masking orders. After several improvements in the past few years [6,3], it includes the verification of most probing-like security notions for different leakage models, including

the robust probing model in the presence of glitches. Its extension into scVerif [20] captures even more advanced hardware side effects [8]. In the same line of work, checkMasks [23] offers the same functionalities with a larger scope (*e.g.*, verification of Boolean to Arithmetic masking conversion) and also a polynomial time verification on selected gadgets. In the same vein, Zhang *et al.* use abstraction-refinement techniques to improve scalability and precision with their tool `SCInfer` [36] whose complexity remains significant. Bordes and Karpman [19] also improves accuracy with the elimination of false negatives.

In a parallel sequence of work, Rebecca [18] was designed to verify the probing security in presence of glitches directly on Verilog implementations. It preceded the similar improvement on maskVerif to also handle hardware implementations. One step further, Coco [29] was designed to check masked software implementations given any possible architectural side effects. Inspired from Rebecca, it analyzes a CPU design as a hardware circuit and investigates all the potential leaks of several shares. Finally, SILVER [33] offers the verification of the classical probing-like security properties for hardware implementations with a complete method based of the analysis of probability distributions.

Since the few past years, the community has made important effort to provide designs in the more realistic random probing model, see *e.g.* [1, 26, 2, 13, 16, 20]. The random probing expandability (RPE) approach developed in [13] currently gives the best complexity to achieve arbitrary random probing security with a constant (and quantifiable) leakage probability. VRAPS [13] was the very first tool to verify random probing properties and it was followed by STRAPS [20], which additionally provides a probabilistic mode to boost the performances, but still does not provide a complete verification method since it uses a set of verification rules from maskVerif which by construction are not complete.

While all existing verification tools struggle to scale either to higher orders or to larger algorithms, a different and complementary approach consists in compiling programs that are secure by design using the composition security properties of some gadgets. This is notably the case of maskComp [6], which tackles the composition problem by introducing the (S)NI notions. This compiler is based on standard t-SNI gadgets (built from the ISW scheme [32]) and inserts t-SNI refresh gadgets at carefully chosen locations for the whole implementation to be t-NI. This was further improved in tightPROVE [15] and tornado [14], which rely on the same gadgets and inserts t-SNI refresh gadgets only when mandatory for the circuit to be t-probing secure.

7 Conclusion

In this paper, we introduce IronMask, a new tool for the formal verification of masking security. Our tool is versatile: it supports the verification of many probing and random probing security, composition and expandability notions. We further introduce a new algebraic characterization for quadratic gadgets with non-linear randomness which notably captures multiplication gadgets refreshing their inputs (which have recently been used in different state-of-the-art masking schemes). From this characterization, we design a complete verification method in the sense that it produces complete verification results (which are notably not affected by false positives). We provide a detailed description of the different algorithms, data structures and optimizations composing our tool. We also introduce a new constructive method for the exhaustive enumeration of so-called incompressible failure tuples. This approach provides speed-up in some cases (specifically for RPS and NI notions).

We developed (in C) and benchmarked the implementation of IronMask, which, enjoying various optimizations, will be made open source. For standard probing security notions (NI, SNI), IronMask has similar performance as existing high-performance tools (maskVerif & matverif), while having the advantage of providing complete results for quadratic gadgets with non-linear randomness. It is also several order of magnitude faster than SILVER, the only other tool providing complete results for those gadgets. For random probing security notions (RPC, RPE) our tool is several orders of magnitude faster than, and also complete compared to, the previous tool VRAPS. These completeness and increased performance allow us to report tighter and better bounds for RPE masking gadgets, which improve the tolerated leakage probability of state-of-the-art random probing secure compilers.

Acknowledgment. This work is partly supported by the French FUI-AAP25 VeriSiCC project.

References

1. 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.
2. Prabhanjan Ananth, Yuval Ishai, and Amit Sahai. Private circuits: A modular approach. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 427–455. Springer, Heidelberg, August 2018.
3. 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.
4. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations. *J. Cryptogr. Eng.*, 10(1):17–26, 2020.
5. 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.
6. 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.
7. Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 535–566. Springer, Heidelberg, April / May 2017.
8. Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR TCHES*, 2021(2):189–228, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8792
9. 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.
10. 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.
11. Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 616–648. Springer, Heidelberg, May 2016.
12. Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 397–426. Springer, Heidelberg, August 2017.

13. Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random probing security: Verification, composition, expansion and new constructions. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 339–368. Springer, Heidelberg, August 2020.
14. Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 311–341. Springer, Heidelberg, May 2020.
15. 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.
16. Sonia Belaïd, Matthieu Rivain, and Abdul Rahman Taleb. On the power of expansion: More efficient constructions in the random probing model. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 313–343. Springer, Heidelberg, October 2021.
17. Sonia Belaïd, Matthieu Rivain, Abdul Rahman Taleb, and Damien Vergnaud. Dynamic random probing expansion with quasi linear asymptotic complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 157–188. Springer, 2021.
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.
19. Nicolas Bordes and Pierre Karpman. Fast verification of masking schemes in characteristic two. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 283–312. Springer, Heidelberg, October 2021.
20. Gaëtan Cassiers, Sebastian Faust, Maximilian Orlt, and François-Xavier Standaert. Towards tight random probing security. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 185–214, Virtual Event, August 2021. Springer, Heidelberg.
21. 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.
22. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.
23. Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 65–82. Springer, Heidelberg, July 2018.
24. Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, Heidelberg, March 2014.
25. 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.
26. 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.
27. 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.
28. 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
29. Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. *IACR Cryptol. ePrint Arch.*, 2020:1294, 2020.
30. Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, Heidelberg, August 1999.
31. Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 567–597. Springer, Heidelberg, April / May 2017.

32. 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.
33. 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.
34. 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.
35. Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.
36. 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.

A Proof of Lemma 2

Proof. The proof follows the different steps of the method described above. All the statements hold from Lemma 1 except that the sets I_1, I_2 are necessary and sufficient for a perfect simulation of $\vec{P}''$. We prove this statement hereafter.

For any random distributions $\mathcal{D}_{\vec{x}_1}$ and $\mathcal{D}_{\vec{x}_2}$ over $\mathbb{K}^n$, we denote $\mathcal{D}_{\vec{P}}$ the distribution induced on $\vec{P}$ by picking $\vec{x}_1 \leftarrow \mathcal{D}_{\vec{x}_1}$, $\vec{x}_2 \leftarrow \mathcal{D}_{\vec{x}_2}$, $\vec{r}_1 \leftarrow \mathbb{K}^{\rho_1}$, $\vec{r}_2 \leftarrow \mathbb{K}^{\rho_2}$. Then I_1 and I_2 are the minimal sets such that for any distributions $\mathcal{D}_{\vec{x}_1}$ and $\mathcal{D}_{\vec{x}_2}$, there exists a probabilistic algorithm $\mathcal{S}$ (the simulator) which given $\vec{x}_1|_{I_1}$ and $\vec{x}_1|_{I_2}$ outputs a tuple $\vec{P}$ which is i.i.d. as $\mathcal{D}_{\vec{P}}$ w.r.t. the random draw $\vec{x}_1 \leftarrow \mathcal{D}_{\vec{x}_1}$, $\vec{x}_2 \leftarrow \mathcal{D}_{\vec{x}_2}$ and the random coins of $\mathcal{S}$.

Direction 1: The sets (I_1, I_2) are necessary to simulate $\vec{P}''$. Here we need to perfectly simulate the distribution of $\vec{P}''$ given the random samplings $\vec{x}_1 \leftarrow \mathcal{D}_{\vec{x}_1}$, $\vec{x}_2 \leftarrow \mathcal{D}_{\vec{x}_2}$, $\vec{r}_1 \leftarrow \mathbb{K}^{\rho_1}$, $\vec{r}_2 \leftarrow \mathbb{K}^{\rho_2}$ for any distributions $\mathcal{D}_{\vec{x}_1}$ and $\mathcal{D}_{\vec{x}_2}$ over $\mathbb{K}^n$. Let us consider the uniform distribution for $\mathcal{D}_{\vec{x}_1}$ then the m first coordinates of $\vec{P}''$ (*i.e.* the expressions of the form (8)) can be written as $\vec{P}''|_{[m]} = (\vec{u}\cdot\vec{h}_{p''_1}, \ldots, \vec{u}\cdot\vec{h}_{p''_m})$, where $\vec{u}$ is a vector uniformly sampled on $\mathbb{K}^{n+\rho_1}$. Recall that the $\vec{h}_{p''_i}$'s corrdinates are expressions of the form (6) w.r.t. $(\vec{x}_2, \vec{r}_2)$.

Given the values taken by the $\vec{h}_{p''_i}$'s we can have different distributions for $\vec{P}''|_{[m]}$. A particular case is the distribution "$\vec{P}''|_{[m]} = \vec{0}$ with probability 1" which appears if and only if $\vec{h}_{p''_1} = \cdots = \vec{h}_{p''_m} = \vec{0}$. In order to evaluate the probability of outputting $\vec{P}''|_{[m]} = \vec{0}$ (which must be exact for a perfect simulation), the simulator must hence evaluate the probability that $\vec{h}_{p''_1} = \cdots = \vec{h}_{p''_m} = \vec{0}$ occurs, which must be further conditioned on the remaining expressions p''_i of the form (6) w.r.t. $(\vec{x}_2, \vec{r}_2)$ (*i.e.* the probes on R_2). This precisely means solving the linear system obtained from the expressions in $\vec{P}_2$ which can be done by Gaussian elimination w.r.t. the $\vec{r}_2$ variables (just as what is actually performed by step 4 of the verification method). The resulting equations without $\vec{r}_2$ variables imply some linear constraints on some of the shares from $\vec{x}_2$. These shares must then be known by the simulator in order to decide if the system has a solution (and to evaluate the probability to get $\vec{h}_{p''_1} = \cdots = \vec{h}_{p''_m} = \vec{0}$). Moreover, these shares are by construction the shares of indexes in I_2.

The exact same proof apply to I_1 by taking the uniform distribution for $\mathcal{D}_{\vec{x}_2}$ and considering the expressions of the form (9) (together with the the probes on R_1).

Direction 2: The sets (I_1, I_2) are sufficient to simulate $\vec{P''}$. Suppose that we can perfectly simulate the tuples of algebraic expressions $\vec{P_1}, \vec{P_2}$ using sets of input shares I_1 on input sharing $\vec{x_1}$ and I_2 on $\vec{x_2}$ respectively as described in step 4 above. Let two new sets of input shares $\tilde{I_1} = [n]$ on $\vec{x_1}$ and $\tilde{I_2} = [n]$ on $\vec{x_2}$.

Observe first that we can perfectly simulate $\vec{P''}$ using the sets of input shares $\tilde{I_1}$ and I_2. In fact, in the algebraic expression of each probe p''_i in $\vec{P''}$ of the form (8), the coordinates of the $\vec{h}^{(p''_i)}$'s can all be perfectly simulated using I_2 since by hypothesis we can perfectly simulate $\vec{P_2}$. Also, the randoms in $\vec{r_1}$ are perfectly simulated by generating uniform random values, and all shares of input $\vec{x_1}$ are simulated using the full input sharing in $\tilde{I_1} = [n]$. Since we can perfectly simulate each term in the expression of p''_i, then we can perfectly simulate the expression p''_i and hence we can perfectly simulate $\vec{P''}$ using $\tilde{I_1}$ and I_2. Similarly, we can perfectly simulate $\vec{P''}$ using the sets of input shares I_1 and $\tilde{I_2} = [n]$ by observing the expressions of p''_i of the form (9).

Thanks to [11, Lemma 7.5] (which demonstrates that if a set of probes can be simulated from different sets of inputs shares, then it can also be simulated by the intersection of these sets), we get that $\vec{P''}$ can be perfectly simulated using the sets of input shares $\tilde{I_1} \cap I_1 = I_1$ and $\tilde{I_2} \cap I_2 = I_2$, which proves that by perfectly simulating the tuples $\vec{P_1}, \vec{P_2}$ using I_1, I_2, we can perfectly simulate $\vec{P''}$ using I_1, I_2. This concludes the proof for this direction.

B Dimension Reduction

We recall hereafter the principle of the dimension reduction optimizations from [11, 19] and show how to make the first one work in the random probing model, and why the second one cannot be used in this model.

Removing elementary probes. Elementary deterministic probes refer to input shares, and products of input shares. The idea behind the removal of those probes is that if a tuple P functionally depends on k input shares, we can always make it depend on $k + k'$ input shares (with $k + k'$ less or equal to the number of shares of the gadget) by adding elementary deterministic probes. For instance, if a tuple t does not depend on the input share a_0, then the tuple (t, a_0) does, and so do any of the tuples of the form $(t, a_0 b_i)$.

The goal of our search procedure, instead of finding tuples of size k_1 that depend on t input shares, now becomes to find tuples of size $k_2 \leq k_1$ that depend on $t - (k_1 - k_2)$ input shares. In the probing model, the existence of such a tuple is enough to know that the property being checked does not hold. In the random probing model, however, we want to generate and count all failures. When we find such a tuple P, we thus generate all expansions of P combined with elementary deterministic probes that leak t input shares, thus making sure that all failures of the gadget are generated.

Similarly, we can remove elementary *random* probes: if, after the Gaussian elimination on a tuple P, some input shares are masked by random variables, we can make them appear by adding the corresponding randoms. For instance, consider the 1-element tuple `a0+r1+a1`. It is easy to see that adding `r1` to this tuple would make it leak `a0` and `a1`. For simplicity, we keep elementary random probes when checking random probing properties, and only perform this optimization for probing properties.

Using reduced sets. This optimization is formally introduced and proven correct in [19]. For completness, we informally recall its principle here.

Let P and P' be two sets of probes. P' is said to be a *reduced set* for P iff $\#P' \leq \#P$ and for every linear combination of probes of P, there exists a linear combination of P' using an equal or lower amount of probes, which contains exactly the same random dependencies, and at least as many input share dependencies. [19] proved that if P is a set of all wires of a gadget G and P' is a reduced set for P, then, to prove that G is NI or SNI, the set of probes P' can be used instead of P to enumerate all tuples. If no failure is found in P', then none can be found in P either, and, conversely, if a failure is found in P', the same failure exists in P.

For instance, if we consider a set $P = \{$ `r0, a0, a1, a0 + r0, a0 + r0 + a1` $\}$, then the set $P' = \{$ `r0, a0, a1, a0 + r0 + a1` $\}$ is a reduced set for P. Evaluating the probing security of a gadget using the latter would yield the same conclusion as with the former, while being faster since the latter contains one less probe.

This optimization is especially potent on ISW-like multiplications, which contain a lot of wires of the form `X + a`$_i$`b`$_j$: the wire `X` can often be omitted since (informally), `X + a`$_i$`b`$_j$ contains the same random dependencies as `X`, but, additionally, contains some additional input shares.

In the random probing model, this optimization cannot be used, because the a set P and a reduced set P' would yield different failures. For instance, consider the sets proposed earlier as example: $P = \{$ `r0, a0, a1, a0 + r0, a0 + r0 + a1` $\}$, and $P' = \{$ `r0, a0, a1, a0 + r0 + a1` $\}$. The tuple `(a0+r0,a0+r0+a1,a1)`, made of wires of P, reveals 2 input shares, and cannot be build from wires of P', since `a0+r0` is not in P'.

C Incompressible tuples

Checking whether a tuple is a failure or not with the SIS procedure is expensive, especially in multiplication gadgets with refreshes on the inputs, where we need to factorize multiplications and perform a total of 3 Gaussian eliminations. An idea from [13] to tackle this issue is to use the notion of *incompressible failure tuple* (Definition 4).

To speed up the verification, we can keep all incompressible failures in a data structure. Then, to check whether a tuple P is a failure, we can start by checking if any subtuple $P' \subset P$ is an incompressible failure. If so, then there is no need to perform the Gaussian eliminations. If no P' is an incompressible failure, then the full verification must be applied. If P is thus found to be a failure, then it is incompressible by definition, and is added to the set of incompressible failures.

For instance, consider the 2-share ISW multiplication of Figure 1a. When considering tuples of size 2, we would find that one of the failures is the tuple P_1 = `(a0*b1+r0+a1*b0, r0)`, since it leaks both shares of `a` (and `b` for that matter). Then, when moving on to tuples of size 3, we would eventually consider the tuple P_2 = `(a0*b1+r0+a1*b0, r0, a0)`. Instead of performing a Gaussian elimination to determine wether P_2 is a failure or not, we could simply observe that it contains P_1, which we already know to be a failure. We can thus conclude that P_2 is a failure as well, without having to perform the Gaussian elimination.

VRAPS [13] uses a list to store the incompressible failures. As a result, the more incompressible failures had been computed, the more expensive membership-checking became. Instead, we use a *trie* (or *prefix tree*). Testing membership of a tuple P of size k inside a trie has complexity $\theta(k)$, and, if P is not in the trie, the search stops strictly before k steps. This is better than a hash table as well, which would require to hash the whole tuple in at least k steps, regardless of whether P is in the hash table or not.

The weakness of this optimization is that for a tuple P of size k, it requires to check all 2^k subtuple $P' \subset P$. This effective complexity can be improved because we use a trie. For instance, if

Table 9: COLUMNS map for the constructive generation of incompressible tuples of Figure 1b

input shares			randoms		
a0	a1	a2	r0	r1	r2
a0	a1	a2	r0	r1	r2
a0 + r0	a1 + r0	a2 + r1	a0 + r0	a2 + r1	a1 + r0 + r2
a1 + r0 + r1	a1 + r0 + r2	a2 + r1 + r2	a1 + r0	a2 + r1 + r2	a2 + r1 + r2
			a0 + r0 + r1	a0 + r0 + r1	

the tuple (p_1, p_2) is not a valid path in the trie, then no tuples (p_1, p_2, p_i) can be in the trie either. Still, the benefits of this approach depend on the size of the gadget, the number of multiplications, and the size of the tuples.

VRAPS [13] is written in SageMath and uses expensive symbolic computation to determine whether tuples are failures or not. As a result, this optimization, albeit implemented with a list instead of a trie, was enough to speed up the verification. In our case, the benefits of this optimization are more situational, and can only be seen on some multiplication gadgets with input refreshes. Still, incompressible tuples are at the core of our new approach to compute failures, introduced in Section 4.5.

D Constructive algorithm

The enumerative approach of Section 4.2 generates a lot of tuples which are trivial non-failures because they do not contain enough shares to be failures, or their shares are masked by random variables. Thus, we designed a constructive algorithm to only generate potential failures. More precisely, our constructive algorithm aims at generating *incompressible failure tuples*, which are defined in [13] as follows:

Definition 4 (Incompressible failure tuple). *A tuple $\vec{P}$ is an incompressible failure tuple if it is a failure, and if no tuple $\vec{P'} \subset \vec{P}$ is a failure itself. ($\subset$ between two tuples means that all wires of $\vec{P'}$ are included in the tuple $\vec{P}$).*

Note: The notion of incompressible failure tuple was used in VRAPS [13] to speed up the enumerative verification of random probing properties. We investiguated this technique, and improved on the implementation of [13] but observed that given the current high performance of our implementation, the optimization from [13] in not advantageous in our case (see Appendix C for more details).

We will start by describing our constructive algorithm for the well-adapted case of LR-gadgets and then explain how to extend it to NLR-gadgets. The idea is that given wires which are all of the form (5), a failure tuple of probes on these wires has a specific form: it contains some wires with input shares, and if those input shares are *masked by randoms*, it contains some additional wires to cancel out those randoms. The expression *masked by randoms* means that the perfect simulation of the considered probe amounts to generating a uniform random value. This is typically the case in a tuple of probes where a random value appears only once in one of the expressions, which then can be used to mask the expression of that probe. To cause a failure event and avoid masking the expressions, additional wires using the same randoms are added to cancel them out.

We start by giving the intuition of the algorithm on the 3-share refresh gadget presented in Figure 1b. We first build a map (Table 9), called COLUMNS, whose keys are the input shares and randoms of the gadget considered, and whose values are all the wires that depend on those inputs and randoms (a wire will be displayed in several columns if it depends on several shares and/or

Algorithm 5 Our constructive algorithm to generate failures. G is the gadget we are considering and n is the number of shares required for a tuple to be a failure.

```
1: procedure UnmaskTuple(G, S, P⃗, unmask_index)
2:     if P⃗ is a failure then
3:         if P⃗ is incompressible then
4:             S ← S ∪ {P⃗}
5:         return
6:     if unmask_index > length(P⃗) then
7:         return
8:     UnmaskTuple(G, S, P⃗, unmask_index+1)
9:     P⃗_Gauss ← GaussElimination(P⃗)
10:    if P⃗_Gauss[unmask_index] contains no randoms then
11:        return
12:    r ← any random from P⃗_Gauss[unmask_index]
13:    for each wire w of G containing r and not in P⃗ do
14:        P⃗' ← P⃗ ∪ w
15:        UnmaskTuple(G, S, P⃗', unmask_index+1)
16: procedure ConstructiveFailuresGenLR(G, n)
17:    S ← ∅
18:    for each tuple P⃗ in ℒ do
19:        UnmaskTuple(G, S, P⃗, 0)
20:    return S
```

randoms). To build a failure tuple that leaks 3 shares, we can pick one wire from each of the `a0`, `a1` and `a2` buckets, say `(a0, a1+r0,a2+r1)` for instance. This tuple is not a failure because the shares `a1` and `a2` are masked by the randoms `r0` and `r1` (the two random values appear only once in the tuple and can be used to mask the corresponding expressions). We thus pick a wire from the `r0` bucket and add it to the tuple, say `a0 + r0 + r1` (which happens to cancel `r1` as well as `r0`). The resulting tuple is `(a0,a1+r0,a2+r1,a0+r0+r1)`, which is a failure. By doing this for every possible wire of each column, we can generate all failures of the gadget of Figure 1b.

Algorithm 5 introduces more formally this procedure for LR-gadgets. This algorithm lists all of the tuples composed of one element from each input share column (line 18); we note the resulting list $\mathcal{L}$. Note that those tuples might have some duplicates since some wires appear in several columns: these duplicates are removed while building the tuples (which implies that the tuples in $\mathcal{L}$ contains possibly less than n elements).

Then, for each tuple in $\mathcal{L}$, the recursive procedure UnmaskTuple adds wires to the tuple so as to cancel the randoms that mask its input shares. This procedure takes at argument the circuit G, the set of incompressible tuples already computed S, a tuple $\vec{P}$ that needs to be turned into a failure and an integer *unmask_index* that contains the next index of $\vec{P}$ that we should try to unmask. First, UnmaskTuple checks if $\vec{P}$ is a failure (line 2). To do so, we can use the procedure SIS_LR (Algorithm 1). If $\vec{P}$ is a failure, we then check if it is incompressible (line 3) by checking if any tuple $\vec{P'} \subset \vec{P}$ is already in S. Ignoring line 8 for now, a Gaussian elimination is then performed on $\vec{P}$ (line 9). If the $unmask\_index^{\text{th}}$ element of the resulting tuple $\vec{P}_{\text{Gauss}}$ contains no random, then there is nothing to unmask, and we can move on to the next index (which was actually already done by line 8). Otherwise, we select any random r of $\vec{P}_{\text{Gauss}}$ and try to add to $\vec{P}$ each wire that contains r (*i.e.*, each wire of the r column of the Columns map, and move to the next *unmask_index* (lines 12 to 15).

As a matter of fact, unmasking each element of $\vec{P}$ one by one misses some failures. This is the reason for line 8, which basically skips the unmasking of the element of $\vec{P}$ at index *unmask_index* to move directly to index *unmask_index*+1. Consider for instance a 2-share gadget and the tuple $\vec{P}$ = `(a0+r0, a1+r1+r0)`. After the Gaussian elimination, the 1st element of $\vec{P}$ is `r0`, and lines 12 to 15 of UnmaskTuple will thus try to add a wire containing `r1` to the tuple. However, this would be missing the fact that the 2nd element of $\vec{P}$, `a1+r1+r0`, already contains `r1` and thus *somewhat* unmasks `a0+r1`. By skipping the first element of $\vec{P}$, we will then try to unmask its second element, by adding the wire `r1` to the tuple for instance. This will produce the tuple `(a0+r0, a1+r1+r0,r1)`, which is a failure, and would have been missed without the recursive call of line 8.

This constructive method is exhaustive since any incompressible failure tuple $\vec{P}$ can be built by taking one elements in each column (and possibly remove duplicates) and then adding necessary elements to remove the masks. More precisely, consider a minimal sub-tuple of $\vec{P}$ which contains one element of each column. This sub-tuple $\vec{P'}$ will be listed in line 8. The other coordinates of $\vec{P}$ are necessary to remove the masks remaining after an application of the Gaussian elimination to $\vec{P'}$ (otherwise $\vec{P}$ would not be incompressible). Since Algorithm 5 is exhaustive in the removal of those masks, it will necessarily build $\vec{P}$ at some point.

Implementation. Our implementation of UnmaskTuple in Algorithm 5 does not perform a full Gaussian elimination at every recursive call. Instead, the elimination is performed on the fly, similarly as we do for the enumerative algorithm (see Section 4.4). Likewise, we keep a variable *input_shares* containing the input shares already revealed by the current tuple $\vec{P}$, which enables to check if $\vec{P}$ is a failure in constant time, without having to call SIS_LR: we can simply check if *input_shares* contains n input shares.

Extension to gadgets with non-linear randomness. The procedure UnmaskTuple of Algorithm 5 only considers gadgets with linear randomness. To adapt it for gadgets with non-linear randomness, we proceed in a similar manner as in SIS_NLR (Algorithm 2): a first step unmasks randoms that are used to refresh outputs, while a second step unmasks randoms that are used to refresh inputs. We call ConstructiveFailuresGenNLR this version of our constructive algorithm and define ConstructiveFailuresGen as the function that chooses between ConstructiveFailuresGenLR and ConstructiveFailuresGenNLR depending on wether its input gadget contains linear or non-linear randomness.

Application. Algorithm 6 shows how to use ConstructiveFailuresGen to compute all failures of a gadget. While the latter returns all incompressible failures, to evaluate the failure function coefficients for random probing notions (RPC, RPE1, RPE2, RPS*), we need to count the number of all failures, regardless of their incompressibility. To do so, we expand all incompressible failures into regular failures by adding wires one by one (using the procedure ExpandTuple, whose pseudo-code is trivial and left out for conciseness). However, doing so will lead to the same tuple being generated multiple times: for instance, if the tuples `(x1, x2)` and `(x1, x3)` are both incompressible failures, expanding them will generate `(x1, x2, x3)` and `(x1, x3, x2)`, which are the same tuple. We thus use a hash table (called S_{failure} in Algorithm 6, and abstracted as a set for simplicity) to store the tuples that we generate and prevent counting multiple times the same tuple. In practice, our hash function returns the sum of the hashes of the indices of wires in the tuple, which results in a fairly low number of collisions.

Algorithm 6 GetCoeffsRPconstr returns an array of c_{max} cells where the k^{th} index contains the number of failure tuples of size k in G

```
procedure GetCoeffsRPconstr(G, c_max)
    coeffs ← empty array
    t ← number of shares in G
    S_incompr ← ConstructiveFailuresGen(G, t)
    S_failure ← ∅
    for k = 1 to c_max do
        S'_failure ← all tuples of S_incompr of size k
        for each tuple P⃗ of S_failure do
            S'_failure ← S'_failure ∪ ExpandTuple(P⃗)
        S_failure ← S'_failure
        coeffs[k] ← number of tuples in S_failure
    return coeffs
```

Remark. We initially tried to count the failures from the incompressibles failures without generating all of them. This problem can be formulated as follows: *Let W be a set of integers (the wires). Let S be a set of subsets of W of arbitrary sizes (the set of incompressible failures). How many subsets of W of size k are super-sets of elements of S (those subsets are non-incompressible failures)?* This is a problem of inclusion-exclusion, and solving it requires computing the intersections of all pairs of sets in the powerset of S. Since there are $2^{|S|}$ such sets, this approach would be prohibitively expensive for any gadget with more than a few incompressible failures.

Limitations. Many tuples are generated multiple times by this constructive algorithms, each time through a different path in the recursion. For instance, on the refresh gadget of Figure 1b, the tuple `(a0,a1+r0,a2+r1,a1+r0+r1)` can be generated by selecting `(a0,a1+r0,a2+r1)` as the initial tuple (line 18 of Algorithm 5), and then adding `a1+r0+r1` at line 14. However, the same tuple can also be generated by selecting `(a0,a1+r0+r1,a2+r1)` line 18 and then adding `a1+r0` line 14. This phenomenon is even more impactful when dealing with multiplication gadgets, because multiple shares of the same input will appear on the same wire, resulting in larger columns (in particular in the "inputs" part of the Columns map), which can lead to a worst complexity than simply enumerating all tuples.

Additionally, when checking properties t-NI with $t < n - 1$ or other properties where a tuple can reveal less than $n-1$ input shares and yet be a failure (*e.g.*, SNI, RPC, RPE), the constructive algorithm is often slower than the traditional enumerative one. Regardless of the property being checked and the failure condition, the constructive algorithm enumerates all tuples. However, with the constructive algorithm, for an n-share gadget, to generate tuples of that leak k shares, all $\binom{n}{k}$ possible combination of shares must be tested. For instance, on our running example of Figure 1b, to generate all failures leaking 2 shares, we would generate all failures leaking the 1^{st} and the 2^{nd} shares, then the ones leaking the 1^{st} and the 2^{nd} share, and, finally, the ones leaking the 2^{nd} and the 3^{rd} share. This formula is also true when checking NI and RP, except that in that case, all n shares must leak, which means that only $\binom{n}{n} = 1$ combination needs to be tested.