PERSEUS – Probabilistic Evaluation of Random probing SEcurity Using efficient Sampling

Sonia Belaïd[1] and Gaëtan Cassiers[1,2]

[1]CryptoExperts, Paris, France
[2]UCLouvain, Louvain-la-Neuve, Belgium

Abstract. Cryptographic implementations are inherently vulnerable to side-channel attacks, which exploit physical leakages such as power consumption. Masking has become the most widely adopted countermeasure to mitigate these threats, as it randomizes intermediate values and makes the leakage less exploitable. Yet, a central challenge remains: how to rigorously assess the concrete security level of masked implementations. To tackle this issue, the random probing model has emerged as a powerful abstraction. It formalizes leakage as random probes in the circuit and, importantly, the security in the noisy leakage model, which closely reflects the behavior of real embedded devices, reduces to security in the random probing model. Hence, proving security in the random probing model provides sound guarantees of practical resistance against side-channel attacks.

Yet, the current state of the art on random probing compilers and verifiers suffers from a clear limitation: scalable approaches yield prohibitively large and inefficient circuits, while tighter methods do not scale to practical circuit sizes. In this work, we bridge this gap by introducing a new methodology that directly estimates the random probing security of large circuits through Monte Carlo sampling, combined with a pruning strategy that drastically reduces the sampling space.

We implement our approach in a new tool, perseus, which supports both gate and wire leakage models. Our experiments demonstrate that perseus can efficiently evaluate masked implementations of AES-128 with $n = 8$ shares, achieving security levels beyond 32 bits, thereby significantly advancing the state of the art in practical verification of side-channel countermeasures.

Keywords: Random Probing Security · Formal Verification · Masking.

1 Introduction

Once deployed on embedded devices, cryptographic algorithms become vulnerable to side-channel attacks [31], which exploit physical emanations to recover secret values. The most widely deployed countermeasure against such attacks is masking [22,26], which splits every sensitive value into n shares so that any subset of them reveals no information about the original value. This approach is justified by the observation of Chari et al. in [22]: since leakages are inherently

noisy, combining the noisy leakages of individual shares makes recovering the original data exponentially harder as the number of shares increases, while the implementation overhead grows at most quadratically [28]. Once all sensitive variables are split into n shares, the main challenge is typically to process operations on these shares in such a way that any set of $n-1$ intermediate variables remain independent from the sensitive values. This is generally achieved by designing small circuits named gadgets that implement basic operations on shared values. For Boolean and arithmetic masking, where the secret is split so that the XOR or the sum of the shares reconstructs the original value, gadgets for linear operations are straightforward: they apply the operation independently to each share. By contrast, gadgets for non-linear operations are more involved, as they require mixing the shares and injecting fresh randomness to prevent leakages from revealing multiple shares at once.

To reason about the security of masking schemes, the community introduced leakage models that formalize the knowledge available to side-channel attackers. Building on the intuition behind masking, the t-probing model of Ishai, Sahai and Wagner [28] assumes that the attacker can observe the exact values of up to t internal variables or wires. This abstraction reflects the increasing difficulty of recovering secrets from multiple independent leakages. While very convenient for proofs and carrying useful intuitions [7], the probing model does not capture more advanced attacks such as horizontal attacks, which exploit the repeated manipulation of sensitive variables. A more realistic framework is the noisy leakage model, introduced by Chari et al. [22] and formalized by Prouff and Rivain [35], which assumes that every operation leaks a noisy function of its internal state. This model reflects the behavior of embedded devices more accurately, but no proof has currently been established directly in this setting without relying on strong assumptions. In 2014, Duc, Dziembowski and Faust [23] provided a reduction from the noisy leakage model to the probing model. Although this reduction is not tight, it relied on an intermediate model: the random probing model, where each variable leaks independently with probability p. Crucially, this model enjoys a tight reduction to the noisy leakage model, making it both theoretically sound and practically tractable.

The first schemes analyzed in the random probing model were proven with a leakage rate p decreasing with the number of shares [28,23,27]. Later, a few constructions were designed to tolerate a constant leakage rate [1,3,2], which more accurately reflects the noise level of real devices. Notably, the approach of Ananth, Ishai, and Sahai [2] was the first to make this assumption explicit, introducing an expansion strategy to derive random probing security from a multiparty computation protocol. Building on this idea, subsequent works [11,14,15] proposed constructions that tolerate even higher leakage rates while reducing complexity. These schemes still rely on the principle of expansion: they start from a base masking scheme with a small number of shares and iteratively apply masking to the already masked circuit until the target security level is reached. The base gadgets must satisfy the property of random probing expandability (RPE), which can be automatically verified for small gadgets using dedicated

tools. Importantly, expansion preserves RPE, which in turn implies threshold random probing composability (threshold RPC), thereby ensuring that the resulting scheme directly achieves random probing security. This framework makes it possible to build circuits achieving arbitrarily high security levels at a fixed noise rate. Yet, the resulting bounds are not tight: after k expansions of a masking scheme with n shares, the circuit contains n^k shares, but the RPE security bound for multiplication scales only as $p^{\lfloor \frac{n+1}{2} \rfloor^k}$, instead of the optimal p^{n^k}.

Another line of works [19,13,16] takes a more direct approach by composing gadgets without using expansion. They introduce alternative simulation-based security properties known as probe distribution tables (PDT) and as (uniformly) cardinal RPC to characterize the security of gadgets, in order to enable tighter composition in the random probing model. Avoiding expansion means that the base gadgets have more shares, which makes the gadget analysis more challenging, requiring new developments in verification tools. Further, the richer gadget characterization makes the composition less scalable. In particular, the cost of computing the security of a circuit is at least exponential in the number of parallel sharings in the circuit (*e.g.* for a block cipher, in the total state and key schedule size), which becomes quickly impractical.

An alternative approach to simulatability, based on leakage diagrams, was proposed in [24,18]. It relies on so-called circular refreshes, inserting linear and multiplication gadgets in between. By characterizing both the computational gadgets and the refresh gadgets, the authors bound the attacker's ability to observe an orbit in the leakage diagrams. Their analysis yields an explicit formula for the compiler as a function of the leakage rate p and the number of shares n. This bound, while insightful, remains relatively coarse and exhibits a non-monotonic behavior in n: it first increases and then decreases, mainly due to the use of a variant of the ISW multiplication [28], which is not robust against horizontal attacks (each share is manipulated n times).

A different direction was explored in [29]: an algebraic approach to random probing security. This method builds on a variant of the multiplication gadget from [8], augmented with refresh gadgets inserted between consecutive gadgets. By analyzing the input–output relationships of each refresh gadget, the leakage that propagates to the surrounding components is quantified. Notably, this work only provides an upper bound on the mutual information between the leakage and a single byte of the secret key, which merely constrains the average leakage.

Most of the above schemes rely on automated verification tools to characterize the individual gadgets. The first such tool in the random probing setting, VRAPS [11], introduced in 2020, was built on the verification rules of maskVerif, an earlier tool for verifying probing security with several shares [5,4]. VRAPS evaluates the RPE and threshold RPC security of gadgets by analyzing all small tuples of wires. For small enough circuits and small p, these tuples dominate the leakage distribution, while larger tuples are unlikely. This makes the computation tractable, since the number of small tuples is polynomial in n, compared to the exponentially many possible tuples. VRAPS was followed by STRAPS [19], which applies a similar strategy to compute PDTs. Like VRAPS,

it exhaustively analyzes small tuples, but it improves the tightness of the bounds by using a probabilistic Monte Carlo algorithm to handle larger tuples, making it more accurate for gadgets with many shares.

Other verification tools focus on improving completeness. Since maskVerif's rules may yield false positives, *i.e.* wire tuples incorrectly classified as depending on secret inputs (and consequently VRAPS and STRAPS inherit this limitation), IronMask [12] was introduced in 2022 to provide exact verification. Its scope is restricted, however, to gadgets involving specific structures (which covers many addition, copy, refresh and multiplication gadgets). More recently, verifMSI [33] has been proposed, following the approach of maskVerif but providing optimized heuristics and implementation, while extending the range of supported circuits. The most recent development is the INDIANA framework of Beierle et al. [9], which provides exact verification for arbitrary masked circuits. INDIANA relies on multi-terminal binary decision diagrams: these are efficient on simple functions but exhibit exponential complexity in the general case, so the approach does not scale well to large circuits or high numbers of shares.

Our Contributions. The current state of the art on random probing compilers and verifiers suffers from a clear limitation: scalable approaches lead to prohibitively expensive masked circuits, while tighter methods cannot efficiently handle large circuits. In this work, we address this gap by introducing a new methodology that directly estimates the random probing security of large circuits via Monte Carlo sampling.

Applying Monte Carlo to this setting raises two main challenges. First, the number of random samples required to verify λ-bit security scales in $\mathcal{O}(2^\lambda)$, making high security levels computationally demanding. Second, for each leaking sample, one must check whether it is independent of the secret, and the cost of this check grows with the circuit size.

To make verification practical, we propose a pruning strategy for Monte Carlo sampling. The key observation is that, in circuits built from gadgets that already verify some security property, the leakage is guaranteed to be independent of the secret for a known subset of the sampled tuples. Our algorithm computes the probability of this subset and samples only from its complement, where failures may actually occur. This pruning drastically reduces the effective sampling space and allows us to bypass the $\mathcal{O}(2^\lambda)$ bottleneck in verifying λ-bit security. We demonstrate the effectiveness of our approach on cryptographic implementations composed entirely of strongly non-interferent (SNI) gadgets, which is a fairly weak restriction: SNI is a widely adopted notion for composable verification in the probing model and efficient SNI gadgets are readily available. Under this assumption, our method discards a large fraction of tuples, thereby making verification significantly faster and more precise.

We implemented our methodology in a new tool, perseus. Beyond its novel approach to random probing security evaluation, perseus integrates its own backend for verifying the independence between leaking tuples and secrets, outperforming the state-of-the-art tools maskVerif and verifMSI. Moreover, perseus supports verification in both the wire random probing model, where each wire leaks with

probability p, and the gate random probing model, where each gate leaks its inputs with probability p. While prior tools have focused exclusively on the wire model, the gate model offers a more realistic abstraction and, crucially, admits a tighter reduction from the noisy leakage model, as recently highlighted in [10].

Finally, we validate our approach by evaluating masked implementations of AES-128 with up to $n = 8$ shares and security levels exceeding 32 bits. Our extensive experiments and comparisons with state-of-the-art techniques confirm substantial performance improvements.

2 Preliminaries

2.1 Notations

For any $n \in \mathbb{N}$, we shall denote $[n]$ the integer set $[n] = [1, n] \cap \mathbb{Z}$. For some tuple $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{K}^n$ and for some set $I \subseteq [n]$, the tuple $(x_i)_{i \in I}$ is denoted $\mathbf{x}|_I$. We use the notation $[\![x]\!] = (x_1, \ldots, x_n) \in \mathbb{K}^n$ for a sharing of a secret value x.

Two probability distributions D_1 and D_2 are *ε-close*, denoted as $D_1 \approx_\varepsilon D_2$, if their statistical distance is upper bounded by ε, namely

$$\mathsf{SD}(D_1; D_2) := \frac{1}{2} \sum_x \left| p_{D_1}(x) - p_{D_2}(x) \right| \leq \varepsilon,$$

where $p_{D_1}(\cdot)$ and $p_{D_2}(\cdot)$ represent the probability mass functions of D_1 and D_2. We write $D_1 \overset{\text{id}}{=} D_2$ to indicate that D_1 and D_2 are *identically distributed*, *i.e.* $p_{D_1}(x) = p_{D_2}(x)$ for all x.

2.2 Arithmetic Circuits

An *arithmetic circuit* over a finite field (or finite ring) $\mathbb{K}$ is a labeled directed acyclic graph whose edges are *wires* and vertices are *arithmetic gates* processing operations on $\mathbb{K}$. The circuit is built from a base set of gates $\mathbb{B} = \{g : \mathbb{K}^\ell \to \mathbb{K}^m\}$, such as addition gates $(x_1, x_2) \mapsto x_1 + x_2$ and multiplication gates $(x_1, x_2) \mapsto x_1 \cdot x_2$. We further consider *probabilistic* (or *randomized*) arithmetic circuits, which also include random gates. Each random gate outputs a fresh value sampled uniformly at random from $\mathbb{K}$. The randomness used by these gates is drawn from a *random tape* denoted by $\rho \in \{0,1\}^*$. A (randomized) arithmetic circuit is formally composed of input gates with fan-in zero and fan-out one, and output gates with fan-in one and fan-out zero. Evaluating an ℓ-input m-output circuit C with a random tape ρ involves writing an input $\mathbf{x} \in \mathbb{K}^\ell$ into the input gates, processing the gates sequentially from inputs to outputs using ρ (only for probabilistic evaluation), and finally reading the output $\mathbf{y} \in \mathbb{K}^m$ from the output gates. This is denoted as $\mathbf{y} = C(\rho, \mathbf{x})$, or simply $\mathbf{y} = C(\mathbf{x})$ since we will always assume ρ to be a uniform random variable over $\{0,1\}^{|\rho|}$.

During the evaluation of a circuit C, each gate computes an internal state determined by its inputs. The tuple of all such gate states is referred to as a

gate assignment of C on input $\mathbf{x}$. Formally, we define an algorithm called the *assign-gates sampler*, denoted $\mathsf{AssignGates}$, which given a (randomized) arithmetic circuit C, a random tape $\rho \in \{0,1\}^*$, a tuple of gate labels $\mathcal{G}$ (a subset of all gate labels in C), and input $\mathbf{x}$, outputs a $|\mathcal{G}|$-tuple $\boldsymbol{g}$, written as $\boldsymbol{g} \leftarrow \mathsf{AssignGates}(C, \rho, \mathcal{G}, \mathbf{x})$. The tuple $\boldsymbol{g}$ corresponds to the exact internal states assigned to the gates labeled by $\mathcal{G}$ during the evaluation of $C(\rho, \mathbf{x})$ and it is deterministic, depending only on C, the randomness ρ, the chosen tuple of labels $\mathcal{G}$ and the input $\mathbf{x}$. Since we are not interested in the specific value of the random tape ρ, we model it as a uniformly distributed random variable over $0, 1^{|\rho|}$. This induces the following probabilistic sampling:

$$\boldsymbol{G} \leftarrow \mathsf{AssignGates}(C, \mathcal{G}, \mathbf{x}).$$

Analogously, we define the *assign-wires sampler* which, given a tuple of wire labels $\mathcal{W}$ (a subset of all wire labels in C), outputs a $|\mathcal{W}|$-tuple $\boldsymbol{w}$ of values assigned to the corresponding wires during the evaluation of C on input $\mathbf{x}$ with uniformly distributed random tape: $\boldsymbol{W} \leftarrow \mathsf{AssignWires}(C, \mathcal{W}, \mathbf{x})$.

2.3 Sharing and Gadgets

In the following, the *n-linear decoding mapping*, denoted by LinDec, refers to the function $\mathbb{K}^n \to \mathbb{K}$ defined, for any $n \in \mathbb{N}$ and $(x_1, \ldots, x_n) \in \mathbb{K}^n$, as

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

We shall further consider that, for every $n, \ell \in \mathbb{N}$, on input $([\![x_1]\!], \ldots, [\![x_n]\!]) \in (\mathbb{K}^n)^\ell$ the n-linear decoding mapping acts as

$$\mathsf{LinDec} : ([\![x_1]\!], \ldots, [\![x_\ell]\!]) \mapsto (\mathsf{LinDec}([\![x_1]\!]), \ldots, \mathsf{LinDec}([\![x_\ell]\!]))$$

Definition 1 (Linear Sharing from [11]). *Let $n, \ell \in \mathbb{N}$. For any $x \in \mathbb{K}$, an n-sharing of x is a random vector $[\![x]\!] \in \mathbb{K}^n$ such that $\mathsf{LinDec}([\![x]\!]) = x$. It is uniform if for any set $I \subseteq [n]$ with $|I| < n$ the tuple $[\![x]\!]|_I$ is uniformly distributed over $\mathbb{K}^{|I|}$. An n-linear encoding is a probabilistic algorithm LinEnc which on input a tuple $\mathbf{x} = (x_1, \ldots, x_\ell) \in \mathbb{K}^\ell$ outputs a tuple $[\![x]\!] = ([\![x_1]\!], \ldots [\![x_\ell]\!]) \in (\mathbb{K}^n)^\ell$ such that $[\![x_i]\!]$ is a uniform n-sharing of x_i for every $i \in [\ell]$.*

In the following, we call an *(n-share, ℓ-to-m) masked circuit*, a randomized arithmetic circuit that maps an input $[\![x]\!] \in (\mathbb{K}^n)^\ell$ to an output $[\![y]\!] \in (\mathbb{K}^n)^m$. A *gadget* denotes typically a small *(n-share, ℓ-to-1) masked circuit* implementing basic arithmetic operations (*e.g.*, addition, multiplication), and we generally view (large) masked circuits as compositions of such gadgets.

When considering wire leakage models, we follow prior work and restrict each computational gate to have fan-out 1.Whenever the output of a gate is used as input by multiple gates, we insert *copy gates* that duplicate their input (with fan-in 1 and fan-out 2). Thus, a gate with fan-out $\beta > 1$ is replaced by an *output replication* subcircuit consisting of $\beta - 1$ copy gates. As a result, each value is no

longer carried by β wires, but instead by $2\beta-1$ wires. When composing gadgets, the output shares of one gadget may serve as input shares for multiple subsequent gadgets. In this case, the gadget is augmented with an output replication circuit: logically, it still produces a single output sharing, but this sharing is duplicated as needed. In the gate leakage model, however, we impose no restriction on gate fan-out, hence we do not use copy gates.

2.4 Security Notions

The most deployed security notions for masked circuits are defined in terms of probes leaking the values of wires. In this work, we instead model probes as leaking gates (revealing their internal states), since, as discussed at the end of this section, this leads to a tighter reduction to practical security.

In a masked circuit C with n shares and ℓ input sharings, a tuple of probes P is secret-independent if their internal states are jointly independent of the input secret x. That is, for any $x, y \in \mathbb{K}^\ell$,

$$\mathsf{AssignGates}(C, P, \mathsf{LinEnc}(x)) \overset{\text{id}}{=} \mathsf{AssignGates}(C, P, \mathsf{LinEnc}(y)).$$

Furthermore, P is simulatable by the input shares $I_1, \ldots, I_\ell$ if there exists a simulator Sim such that for any $[\![x]\!] \in (\mathbb{K}^n)^\ell$,

$$\mathsf{Sim}(C, P, (\boldsymbol{x_1}|_{I_1}, \ldots, \boldsymbol{x_\ell}|_{I_\ell})) \overset{\text{id}}{=} \mathsf{AssignGates}(C, P, (\boldsymbol{x_1}, \ldots, \boldsymbol{x_\ell})).$$

Probing security [28] is the standard notion to assess the security of masked circuits: a circuit is t-probing secure if any set of t probes is secret-independent. To enable composition, the notion of Strong Non-Interference (SNI) was later introduced [6]: an n-share gadget is t-SNI if any set of t_i internal probes and t_o output shares can be perfectly simulated from at most t_i shares of each input, as long as $t_i + t_o \leq t$. This guarantees that composing SNI gadgets yields a $(n-1)$-probing secure circuit: probes of one gadget can be simulated from its input shares, which are outputs of previous gadgets, and these inherited probes can then be simulated for free.

Definition 2 (Strong Non-Interference ([6], adapted to gate leakage)). *Let C be an (n-share, ℓ-to-1) masked circuit. C is t-Strong Non-Interferent (t-SNI), if for every tuple P of probes of C such that $|P| \leq t_i$, and every set $J \subseteq [n]$ such that $|J| \leq t_o$ and $t_i + t_o \leq t$, there exists a two-stage simulator $\mathsf{Sim} = \big(\mathsf{Sim}_1, \mathsf{Sim}_2\big)$ such that for every input $([\![x_1]\!], \ldots, [\![x_\ell]\!]) \in (\mathbb{K}^n)^\ell$,*

- $\mathsf{Sim}_1(P, J) = (I_1, \ldots, I_\ell)$ *where* $I_1, \ldots, I_\ell \subseteq [n]$ *and* $|I_1|, \ldots, |I_\ell| \leq |P|$,
- $\mathsf{Sim}_2(P, J, ([\![x_1]\!]|_{I_1}, \ldots, [\![x_\ell]\!]|_{I_\ell})) \overset{id}{=} \big(\mathsf{AssignGates}(C, P, ([\![x_1]\!], \ldots, [\![x_\ell]\!])), [\![y]\!]|_J\big)$

where $[\![y]\!] = C([\![x_1]\!], \ldots, [\![x_\ell]\!])$. An n-share gadget is simply said to be SNI *if it is $(n-1)$-SNI.*

A significant limitation of probing security is that the threshold t is independent of the size of the circuit, whereas in practice, evaluating a larger circuit causes more leakage. The region probing model, introduced in [6], solves this limitation by partitioning the circuit into regions, tolerating t probes in each region. A circuit is t-region probing secure for a partition $\mathcal{G}_1, \ldots, \mathcal{G}_k$ of its gates if any tuple of probes P such that $|P \cap \mathcal{G}_i| \leq t$ is secret-independent.

The random probing model was introduced to better capture real-world leakage. Let $p \in [0, 1]$ denote a constant leakage probability parameter, also referred to as the *leakage rate*. In the (gate) p-random probing model, the evaluation of an arithmetic circuit C leaks the internal state of each gate with probability p, where all gate leakage events are mutually independent. Formally, we introduce the *leaking-gates sampler* as the probabilistic algorithm that outputs a tuple $\mathcal{G}$ of gate labels from C, where each label is independently selected with probability p:

$$\mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p).$$

We denote by $\mathcal{T}_C$ the set of all finite tuples over the gates of C. In particular, any output of the sampler satisfies $\mathcal{G} \in \mathcal{T}_C$. Analogously, we define the *leaking-wires sampler*, denoted $\mathcal{W} \leftarrow \mathsf{LeakingWires}(C, p)$, which outputs a tuple $\mathcal{W}$ of wire labels rather than gate labels.

Definition 3 (Random Probing Security ([11], adapted to gate leakage)). *Let $\ell \in \mathbb{N}$ and $p, \varepsilon \in [0, 1]$. A randomized arithmetic circuit C is* (p, ε)-random probing secure *(or RPS for short) with respect to encoding* LinEnc *if there exists a simulator* Sim *such that for every input $\mathbf{x} \in \mathbb{K}^\ell$, the distribution of $\mathsf{Sim}(C)$ is ε-close to the distribution of $\mathsf{AssignGates}(C, \mathcal{G}, \mathsf{LinEnc}(\mathbf{x}))$ where $\mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p)$.*

The parameter ε, referred to as the simulation failure probability, directly bounds the distinguishing advantage of any adversary trying to distinguish between the real leakage and the simulated leakage. We equivalently define (p, ε)-random probing security for a circuit in the wire leakage setting, where we use $\mathsf{LeakingWires}$ and $\mathsf{AssignWires}$ instead of $\mathsf{LeakingGates}$ and $\mathsf{AssignGates}$.

In [10], the methodology for obtaining practically secure cryptographic implementations on microcontrollers relies on random probing security in the gate model, since the leakage of a processor operation depends on all its operands. While the authors show in their Lemma 1 that if a circuit with two-input gates is (p, ε)-random probing secure in the wire setting, then it is also (p^2, ε)-secure in the gate setting, this reduction is not tight. Consequently, evaluating security directly in the gate model yields tighter bounds, whenever possible.

3 Direct Approach

Our goal in this work is to empirically estimate the random probing security level of a circuit C at leakage rate p (see Definition 3). To this end, we perform Monte Carlo sampling: we draw N leakage tuples $(\mathcal{G}_1, \ldots, \mathcal{G}_N)$ i.i.d. from

$\mathsf{LeakingGates}(C, p)$, where each gate is included independently with probability p. For each $\mathcal{G}_i$, we check whether it is secret-independent using a procedure denoted $\mathsf{CheckIndep}$. Whenever this is not the case, we increment a counter c. Consequently, c follows a Binomial distribution $\mathcal{B}(N, \varepsilon)$, where ε denotes the probability $\mathsf{CheckIndep}$ returns $\perp$ (*i.e.* the leakage might depend on secrets), and thus relates to the random probing security of C. From the observed value of c, we derive a $(1-\delta)$-upper bound for ε, ensuring that the circuit C is (p, ε)-random probing secure with confidence level $(1-\delta)$. The complete procedure is given in Algorithm 1, where $I^{-1}(\alpha; a, b)$ is the inverse of the regularized incomplete beta function with parameters (a, b), *i.e.* the α-quantile of the $\mathrm{Beta}(a, b)$ distribution. Its correctness is established in Lemma 1, which essentially recovers the Clopper–Pearson confidence interval [37], as in [19]. For completeness, we provide a short proof below.

Algorithm 1 Direct Monte Carlo for Random Probing Security

Input: Circuit C; random probing probability $p \in [0, 1]$; confidence level $1-\delta$; number of samples N.

Output: Upper bound ε^U such that C is (p, ε)-RPS for some $\varepsilon \in [0, \varepsilon^U]$ with probability at least $1 - \delta$; lower bound ε^L such that $\varepsilon^L \leq \Pr[\mathsf{CheckIndep}(\mathsf{LeakingGates}(C, p), C) = \perp]$ with probability at least $1 - \delta$.

```
c ← 0
for i = 1, ..., N do
    G ← LeakingGates(C, p)
    if ¬CheckIndep(G, C) then
        c ← c + 1
if c = N then ε^U ← 1 else ε^U ← I^{-1}(1 − δ; c + 1, N − c)
if c = 0 then ε^L ← 0 else ε^L ← I^{-1}(1 − δ; c, N − c + 1)
```

Lemma 1. *Let $p, \delta \in [0, 1]$ and $N \in \mathbb{N}$. Let C be a randomized arithmetic circuit. The bounds $(\varepsilon^L, \varepsilon^U)$ returned by Algorithm 1 on inputs C, p, $1-\delta$, and N satisfy*

$$\Pr[\varepsilon^U \geq \varepsilon'] \geq 1 - \delta \quad \text{and} \quad \Pr[\varepsilon^L \leq \varepsilon'] \geq 1 - \delta \tag{1}$$

where $\varepsilon' = \Pr[\mathsf{CheckIndep}(\mathsf{LeakingGates}(C, p), C) = \perp]$.

Proof. We aim to derive a sound and tight upper bound ε^U on ε' that satisfies Equation 1. To proceed, let us define $\varepsilon^U : \mathbb{N} \to [0, 1]$ as an increasing function to represent the bound ε^U for all possible values of c and $c^i_\varepsilon = \min_{c \in \mathbb{N}}\{c : \varepsilon^U(c) \geq \varepsilon\}$. By construction, we have

$$\varepsilon^U(c) \geq \varepsilon' \Leftrightarrow c \geq c^i_\varepsilon,$$

which implies

$$\Pr[\varepsilon^U(c) \geq \varepsilon'] = \Pr[c \geq c^i_\varepsilon] = 1 - F(c^i_\varepsilon; N, \varepsilon'),$$

where $F(k; N, p)$ denotes the cumulative distribution function (CDF) of the binomial distribution, *i.e.* $F(k; N, p) = \Pr[X \leq k]$ for $X \sim \mathcal{B}(N, p)$. The function ε^U can be defined such that $F(c; N, \varepsilon^U(c)) = \delta$ which satisfies the condition of Equation 1. Finally, we remark that the binomial CDF can be expressed using the regularized incomplete beta function $I_x(a, b)$. Specifically:

$$F(c; N, \varepsilon^U(c)) = I_{1-\varepsilon^U(c)}(N - c, c + 1) = 1 - I_{\varepsilon^U(c)}(c + 1, N - c),$$

and therefore

$$\varepsilon^U(c) = \begin{cases} 1 & \text{if } c = N \\ I^{-1}(1 - \delta; c + 1, N - c) & \text{otherwise.} \end{cases}$$

A similar reasoning gives $\varepsilon^L(c) = I^{-1}(1-\delta; c, N-c+1)$ for $c \neq 0$, and $\varepsilon^L(c) = 0$ if $c = 0$. □

Corollary 1. *If, for any $\mathcal{G} \in \mathcal{T}_C$, $\mathsf{CheckIndep}(\mathcal{G}, C) = \top$ implies that $\mathcal{G}$ is secret-independent, then there exists ε such that C is (p, ε)-RPS and $\Pr[\varepsilon^U \geq \varepsilon] \geq 1-\delta$.*

Proof. Let $\varepsilon = \Pr[\mathsf{CheckIndep}(\mathsf{LeakingGates}(C, p), C) = \bot]$, the hypothesis implies that C is (p, ε)-RPS. □

Therefore, if CheckIndep is correct (*i.e.* for every $\mathcal{G}$ that is not secret-independent, we have $\mathsf{CheckIndep}(\mathcal{G}, C) = \bot$), then ε^U is a valid upper bound on the random probing security level. The tightness of this bound depends on two components: (i) the gap due to the Monte Carlo bound, and (ii) the false-positive rate of CheckIndep (*i.e.* how often $\mathsf{CheckIndep}(\mathcal{G}, C) = \bot$ while $\mathcal{G}$ is actually secret-independent). For (i), the difference between ε^U (Monte Carlo upper bound) and ε' (the true probability that CheckIndep returns $\bot$) is well controlled: with probability at least $1 - 2\delta$, $\varepsilon' \in [\varepsilon^L, \varepsilon^U]$, and the size of this interval converges to 0 as N tends to infinity. Due to (ii), however, ε^L cannot be taken as a lower bound on ε; it only indicates the tightness of (i). The loss of tightness from (ii) is harder to quantify, as we have no matching lower bound. We discuss this issue further in the next section, together with the implementation of CheckIndep.

3.1 Checking dependence on secrets

We now turn to the implementation of CheckIndep, whose goal is to decide whether a tuple of wires $\mathcal{W}$ is secret-independent. When checking a set of gates $\mathcal{G}$, $\mathcal{W}$ is taken as the tuple of all their input wires. This problem has been extensively studied in the context of probing security verification.

A naive solution is to enumerate all possible assignments to the input and random wires and compare the leakage distributions. This provides an exact answer but grows exponentially with the number of these wires and is therefore impractical for all but the smallest circuits. Improvements over this approach exploit the structure of the considered (Boolean) functions to optimize the computations, but remain limited to small circuits either due to poor scaling with

the circuit size (*e.g.* SILVER [30]) or because they handle only specific circuit structures (*e.g.* IronMask [12]). Given our goal of verifying large circuits, such tools are not suitable for implementing CheckIndep.

Algebraic approaches such as maskVerif [5] or verifMSI [33] scale to larger designs but are not complete: they may produce *false positives* (failing to prove independence when it actually holds), leading to part (ii) non-tightness. In practice, these tools can verify large classes of circuit for t-probing security (as well as asymptotically tight bounds in the random probing model for STRAPS [19]), which shows that this approach is sufficient to verify many tuples from practical masked implementations.

In this work, we first consider using maskVerif to implement CheckIndep. This tool represents circuits in an abstract ring with addition and multiplication operations, as a graph of expressions. For verifying secret-independence, we encode the circuit $C(\mathsf{LinEnc}(\mathbf{x}))$ and mark the probed wires. maskVerif then performs simplifications on this graph, based either on algebraic identities, or on the "random" rule: if a random input r is not directly probed and appears only once in the graph, in an expression of the form $r + e$ (where e is any expression), then $r + e$ is replaced with r (this does not change the distribution of the probes). Further, it also supports opaque gates which to not support algebraic simplifications but benefit from the random rule if they are bijective: for a bijective gate b, $b(r)$ can be replaced by r.

We also consider the use of verifMSI, which is based on similar principles as maskVerif, but has a richer representation of field elements with bit vectors. Then, simplifications can use either algebraic rules from the finite field in which the operations are defined, or use a Boolean view of the operations.

Finally, we implemented our own CheckIndep, named favom (for Fast Verification of Masking). It essentially re-implements maskVerif's algorithm with different data-structure choices to gain speed. In a nutshell, we optimize memory usage and avoid allocating memory individually for each node in an expression graph. We also optimize the book-keeping of expression's children (*i.e.* the list of expressions in which a given sub-expression appears) by keeping a count of children instead of the complete list. This algorithm runs in worst-case time $\mathcal{O}(|C|^2)$ (which corresponds to applying the random rule to every node in the circuit), with a significantly lower average case (there are typically less than a dozen rule applications per probe before simplification finishes, *i.e.* linear time in practice). While implementing favom, we reviewed the code of maskVerif and identified an algorithm with (accidental) exponential complexity in the circuit depth. After fixing this issue, the asymptotic complexity of maskVerif's verification matches that of favom, yet in practice it remains about 15 times slower.

3.2 Computational cost

The computational cost of Algorithm 1 is driven by the number of samples N, since the two main expensive steps are executed once per sample: (i) drawing a leakage tuple from $\mathsf{LeakingGates}(C, p)$ and (ii) checking its dependency with

the secrets. Let T_{samp} and T_{ver} denote the per-iteration costs of these steps; the overall runtime is in $\mathcal{O}\big(\big(N\,(T_{\text{samp}} + T_{\text{ver}})\big)$, typically dominated by T_{ver}.

Sampling LeakingGates. Sampling from LeakingGates is fairly easy: include each gate independently with probability p (*i.e.* sample from a Bernoulli distribution of parameter p for each gate). For small p, a more efficient alternative is to first draw the number of leaking gates from a Binomial distribution: $n_g \sim \mathcal{B}(|C|, p)$ (where $|C|$ is the number of gates in C), then sample n_g distinct gates uniformly at random. This reduces the expected cost T_{samp} from $\mathcal{O}(|C|)$ to $\mathcal{O}(p|C|)$.

Estimating N. The tightness of the bounds $(\varepsilon^L(c), \varepsilon^U(c))$ can be characterized in terms of the number of "bits of security" between the bounds: $\log_2(\varepsilon^U(c)/\varepsilon^L(c))$, with the worst-case occurring when $c = 0$ and hence $\varepsilon^L(c) = 0$. Using the identity $I_x(a, 1) = x^a$ and the approximation $\ln(1 - \varepsilon^U(c)) \approx -\varepsilon^U(c)$, the upper bound simplifies to $\varepsilon^U(0) \approx \frac{\ln(1/\delta)}{N}$. In this worst-cast, verifying that a circuit is $(p, 2^{-\lambda})$-random probing secure (*i.e.* a λ-bit security level) means obtaining $\varepsilon^U(0) \leq 2^{-\lambda}$, which requires

$$N \approx 2^{\lambda} \ln(1/\delta)$$

Monte Carlo samples.

For instance, on an 8-core laptop processing (sampling and verifying) 2^8 sampled tuples per second per core (*i.e.* $T_{\text{samp}} + T_{\text{ver}} \approx 2^{-8}$ seconds), we can collect enough samples in about 17 h to obtain a bound of 20 bits of security at confidence $\delta = 10^{-3}$. On the high-performance end, a budget of 10 000 $ on AWS cloud machines[1] yields roughly 2^{31} core·hours, which, under the same per-iteration cost assumptions, suffices to target ≈ 32 bits. This shows that claiming $\lambda = 32$ bits of security is computationally feasible, albeit costly. This *brute-force* Monte Carlo approach is thus practical when targeting intermediate security bounds, *e.g.* when 2^{λ} only needs to bound the number of traces a realistic side-channel adversary might collect. However, it has limitations: it only provides a lower bound for a fixed value of p, and remains computationally expensive.

4 Pruning the Leakage Distribution

In this section, we improve the basic estimator of a circuit's random probing security level described in Section 3. We keep the Monte Carlo procedure involving the sampling of gate tuples and their verification, but introduce a key idea: *pruning*. We bypass a carefully identified subset of tuples whose dependency with the secrets is known *a priori*, and thus does not need to be verified. This reduces the number of expensive verification calls needed to reach a given security/confidence target; although the sampling scheme becomes slightly more involved, the overall cost decreases because the effective sample size N can be chosen significantly smaller.

[1] `https://aws.amazon.com/ec2/spot/pricing/` (September 2025)

4.1 Main Idea

Let C be a randomized arithmetic circuit composed of a set of gadgets, where each gadget produces a single output sharing that may serve as input to multiple subsequent gadgets. Suppose we can identify a subset of tuples $\mathcal{S}_G \subset \mathcal{T}_C$ whose inputs are all jointly independent on C's secret inputs and that are selected by the leaking-gates sampler $\mathsf{LeakingGates}(C, p)$ with known probability

$$\Pr[\mathcal{G} \in \mathcal{S}_G \mid \mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p)].$$

Then, we can restrict the Monte Carlo verification to tuples in $\mathcal{T}_C \setminus \mathcal{S}_G$ and rescale the resulting estimate by a factor of $\alpha = \Pr[\mathcal{G} \notin \mathcal{S}_G \mid \mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p)]$. This leads to a substantial gain in efficiency when α is close to 0. This optimization is formalized in Algorithm 2, which computes a statistical upper bound ε^U on the simulation failure probability in the (p, ε)-random probing model, with confidence level $1 - \delta$.

Algorithm 2 Improved Monte Carlo for Random Probing Security

Input: Circuit C; random probing probability $p \in [0, 1]$; confidence level $1-\delta$; number of samples N.

Output: Upper bound ε^U such that C is (p, ε)-RPS for some $\varepsilon \in [0, \varepsilon^U]$ with probability at least $1 - \delta$; lower bound ε^L such that $\varepsilon^L \leq \Pr[\mathsf{CheckIndep}(\mathsf{LeakingGates}(C, p), C) = \bot]$ with probability at least $1 - \delta$.

$(\alpha^L, \alpha^U) \leftarrow \mathsf{EstimateProbNotInSG}(C, p)$ ▷ $\alpha^L \leq \Pr[\mathcal{G} \notin \mathcal{S}_G \mid \mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p)] \leq \alpha^U$
$c \leftarrow 0$
for $i = 1, \ldots, N$ **do**
  $\mathcal{G} \leftarrow \mathsf{SampleOutSG}(C, p)$ ▷ Sample $\mathcal{G} \leftarrow \mathsf{LeakingGates}(C, p)$ such that $\mathcal{G} \notin \mathcal{S}_G$
  if $\neg\mathsf{CheckIndep}(\mathcal{G})$ **then**
    $c \leftarrow c + 1$
if $c = N$ **then** $\varepsilon^U \leftarrow \alpha^U$ **else** $\varepsilon^U \leftarrow \alpha^U I^{-1}(1 - \delta; c + 1, N - c)$
if $c = 0$ **then** $\varepsilon^L \leftarrow 0$ **else** $\varepsilon^L \leftarrow \alpha^L I^{-1}(1 - \delta; c, N - c + 1)$

While conceptually simple, this approach relies on the efficient design of two new key components:

- $\mathsf{EstimateProbNotInSG}$ computes bounds on the probability α that a sampled tuple does not lie in $\mathcal{S}_G$,
- $\mathsf{SampleOutSG}$ samples uniformly in the remaining tuples $\mathcal{T}_C \setminus \mathcal{S}_G$.

Before diving into the details of these two components, the next section explains how to build a strategically chosen subset $\mathcal{S}_G$ that minimizes α, while ensuring that the remaining tuples can still be sampled efficiently.

4.2 Identification of Simulatable Tuples in $\mathcal{S}_G$

The first step of our methodology is to identify a subset of tuples in $\mathcal{T}_C$ that are known to be simulatable (without using $\mathsf{CheckIndep}$), and for which the

leakage probability can be tightly estimated. We begin by discussing the main security properties typically satisfied by individual gadgets and explain our focus on implementations built exclusively from SNI gadgets. We then specify the SNI constraints and characterize the conditions under which a tuple can be immediately considered simulatable.

Structured Implementations. Most masked implementations are built from small gadgets that satisfy well-established local security properties. The most basic guarantee comes from probing security: any tuple of at most t probes that is entirely contained within a single t-probing secure gadget can be simulated without knowledge of the secret inputs. However, this approach is inherently local and scales poorly with the circuit size, especially under realistic leakage probabilities where the attacker may observe probes across multiple gadgets. A stronger structural criterion is offered by t-region probing security, which partitions the circuit into distinct regions, each of which tolerates up to t probes. Any tuple that contains fewer than t probes in each region is simulatable by definition and can thus be safely excluded. However, the effectiveness of this approach strongly depends on the probability that any individual region will be violated.

Concretely, if a gadget G contains $|G|$ internal gates, the probability that it leaks more than t gates under a random probing model with leakage probability p is given by:

$$q = \sum_{i=t+1}^{|G|} \binom{|G|}{t+1} p^i (1-p)^{|G|-i}.$$

If the circuit contains enough gadgets, it becomes likely that at least one of them will exceed the t-probe threshold. For instance, the multiplication gadget from [28] (named ISW) with n shares contains $|G| = 3n^2 - 2n$ gates and satisfies t-region probing security for $t = \lfloor (n-1)/2 \rfloor$. Taking $n = 8$ and $p = 2^{-10}$, the probability that a single gadget exceeds t probes is around 3×10^{-5}. Considering an implementation of AES in $\mathbb{F}_{256}$ which uses 640 multiplication gadgets, the overall probability of violation raises to 2% (this ignores the other $\approx$ 2k gadgets), which represents a gain of only $50\times$ over the direct method for the value of the worst-case RPS bound. This fairly high probability of exceeding the threshold is in part caused by the value of t. Indeed, the bound $t < n/2$ is intrinsic to the region probing model: a sharing which is the output of a region is also an input of another region, therefore the adversary may probe $2t$ shares of this sharing.

This limitation motivates us to raise the threshold by considering a more fine-grained structural property, namely the t-SNI property (Definition 2), which is verified by many non-linear gadgets from the literature with $t = n - 1$.[2] The high-level idea is the following: instead of partitioning the circuit in regions, we consider larger overlapping subsets of the circuit that we call dependency groups.

[2] We consider gate leakage, while most works on t-SNI consider wire leakage. However, gate leakage is a weaker leakage model than glitch-robust leakage, for which the ISW multiplication is $(n-1)$-SNI [25].

Then, if there are at most $n-1$ probes in each dependency group, we know that the probes are jointly simulatable.

Simulation using SNI constraints. More precisely, dependency groups are composed of a gadget and its output group, defined as follows:

Definition 4 (Output Group). *Let C be a randomized arithmetic circuit composed of a set of gadgets $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$, where each gadget produces a single output sharing that may be used as input by multiple subsequent gadgets. For any gadget $G_i \in \mathcal{G}$, we define its* output group*, denoted OG_i, as the multiset of indices j such that an input sharing of G_j is the output sharing of G_i, and the multiplicity of G_j in OG_i is the number of input sharings of G_j which are the output sharing of G_i.*

The significance of these groups lies in the following lemma, which allows us to systematically discard many tuples from the verification set:

Lemma 2. *Let C be a randomized arithmetic circuit composed of k gadgets $\{G_1, G_2, \ldots, G_k\}$. Assume that each gadget G_i satisfies the t-SNI gate-leakage property. Let $\mathcal{G} \in \mathcal{T}_C$ be a tuple of gates, and denote by $\mathcal{G}_i \subseteq \mathcal{G}$ the subset of gates belonging to gadget G_i for $i = 1, \ldots, k$. We define, for each i,*

$$\Delta_i = \begin{cases} \top & \text{if } |\mathcal{G}_i| + \sum_{j \in \mathsf{OG}_i} |\mathcal{G}_j| > t \\ \bot & \text{otherwise} \end{cases}$$

which represents a violation of the SNI constraint on G_i. Consequently, we define $\mathcal{S}_G$ as

$$\mathcal{G} \in \mathcal{S}_G \equiv \bigwedge_{i=1}^{k} \neg \Delta_i, \tag{2}$$

and then $\mathcal{G} \in \mathcal{S}_G$ implies that $\mathcal{G}$ is secret-independent.

Proof. We work by induction on the circuit from outputs to inputs (*i.e.* from $\{G_k\}$ to $\{G_1, \ldots, G_k\}$), showing that the leakage of all gadgets can be simulated using for each input sharing of the circuit a number of shares at most $\sum_{j \in J} |\mathcal{G}_j|$, where J is the multiset of gadgets using that input sharing (with multiplicities counting number of uses). The base case $\{G_k\}$ is straightforward: given $\neg\Delta_k$ (*i.e.* the SNI constraint is not violated with $\mathcal{G}_k$) and given that $\mathsf{OG}_k = \emptyset$, we can use the t-SNI simulator of G_k to simulate the leakage, using $|\mathcal{G}_k|$ shares of each input sharing.

Then, we consider the induction step, from $\{G_{i+1}, \ldots, G_k\}$ to $\{G_i, \ldots, G_k\}$. By induction hypothesis, we need to simulate at most $\sum_{j \in \mathsf{OG}_i} |\mathcal{G}_j|$ output shares of G_i. Therefore, thanks to $\neg\Delta_i$, we can use the t-SNI simulator to simulate those shares as well as the leakage in G_i. We then use the simulator from the induction hypothesis for the remaining gadgets. For each input sharing of the circuit $\{G_i, \ldots, G_k\}$, the bound on the number of shares from the induction hypothesis is trivially satisfied if the sharing is not an input of G_i. Otherwise, the simulation of G_i requires at most $|\mathcal{G}_i|$ additional shares, which still satisfies the induction hypothesis. □

This structural insight allows us to prune a substantial portion of the tuple space $\mathcal{T}_C$ before performing any costly simulation, thereby enabling efficient and scalable verification. In the remainder of this paper, we focus on this t-SNI setting, which captures many existing non-linear gadgets and already enables significant reductions of the verification space.

We note, however, that more fine-grained results could be achieved by characterizing the simulation capabilities of each gadget individually. For instance, if a tool precomputes, for each gadget, which output shares are simulatable along with the internal leakage from which set of (or how many) input shares, one could further reduce the verification effort. We leave the design of such advanced characterizations to future work, as we demonstrate in this paper that even with simpler and well-known properties like SNI, our methodology can yield substantial efficiency gains.

4.3 Computation of α (EstimateProbNotInSG)

We now present our approach to estimate the parameter α, defined as the probability that a leaking tuple of gates $\mathcal{G}$ cannot be simulated from the underlying SNI structure of the circuit (or equivalently that $\mathcal{G} \notin \mathcal{S}_G$).

Let C be a circuit composed of gadgets $G_1, G_2, \ldots, G_k$, each satisfying the t-SNI gate-leakage property. Let $\mathcal{G}$ be a random variable representing a tuple of leaking gates sampled from $\mathsf{LeakingGates}(C, p)$ for some $p \in [0, 1]$, with $\mathcal{G}_i \subseteq \mathcal{G}$ the leaking gates belonging to gadget G_i. In theory, we can compute $\alpha = \Pr[\mathcal{G} \notin \mathcal{S}_G]$ exactly by summing over all possible leakage configurations, as follows:

$$\alpha = 1 - \sum_{\gamma_1=0}^{|G_1|} \cdots \sum_{\gamma_k=0}^{|G_k|} \prod_{i=1}^{k} \Pr[|\mathcal{G}_i| = \gamma_i] \cdot \mathbf{1}\left[\gamma_i + \sum_{j \in \mathsf{OG}_i} \gamma_j \leq t\right],$$

where $|\mathcal{G}_i|$ follows a binomial distribution:

$$\Pr[|\mathcal{G}_i| = \gamma_i] = \binom{|G_i|}{\gamma_i} p^{\gamma_i} (1-p)^{|G_i| - \gamma_i}.$$

This approach has exponential complexity in k and quickly becomes intractable for large circuits. To overcome this, we apply the inclusion–exclusion principle to Equation 2, which yields bounds for α that are easier to compute.

Lemma 3. *Let C be a circuit composed of gadgets $G_1, G_2, \ldots, G_k$, each satisfying the t-SNI gate-leakage property. Let $\mathcal{G}$ be a random variable representing a tuple of leaking gates sampled from $\mathsf{LeakingGates}(C, p)$ for some $p \in [0, 1]$. Then, for $1 \leq s \leq k$, we have*

$$B_{2s} \leq \alpha \leq B_{2s+1}$$

where

$$B_s = \sum_{i=1}^{s} (-1)^{i+1} \cdot T_i \quad \textit{with} \quad T_i = \sum_{J \subseteq [\![k]\!], |J|=i} \Pr\left[\bigwedge_{j \in J} \Delta_j\right]. \tag{3}$$

Proof. From Equation 2, we get a new expression for α:

$$\alpha = \Pr\left[\bigvee_{i=1}^{k} \Delta_i\right].$$

Applying Bonferroni inequalities to this expression gives the expected result. □

If p is sufficiently small, then T_i will decrease quickly when i grows, since it corresponds to increasingly large tuples of probes. Notably, in our experiments of Section 5, computing B_2 (that we denote α^L) or B_3 (that we denote α^U) give sufficiently tight bounds on α. It remains to show how to efficiently compute $\Pr\left[\bigwedge_{j\in J} \Delta_j\right]$ for a given J, and then T_i.

Let $J' = J \cup \bigcup_{j\in J} \mathsf{OG}_j$ be the set of all gadgets involved in computing Δ_j. Define the joint space:

$$R_j = \{0, \ldots, |G_j|\}, \quad R_{J'} = \prod_{j\in J'} R_j$$

where $|G_j|$ is the number of gates in G_j. We expand the joint probability by total probability:

$$\Pr\left[\bigwedge_{j\in J} \Delta_j\right] = \sum_{(\gamma_j)_{j\in J'}\in R_{J'}} \left(\prod_{j\in J} \delta_j(\gamma) \cdot \prod_{j\in J'} \Pr[|\mathcal{G}_j| = \gamma_j]\right),$$

where $\delta_j(\gamma) = 1$ if $\gamma_j + \sum_{\ell\in\mathsf{OG}_j} \gamma_\ell > t$, and 0 otherwise. This computation involves a sum-product form for which many variable elimination algorithms exist. The complexity is exponential in the treewidth of the graph, but if we can keep $|J|$ small, the graph will have overall few factors (up to $|J|$), so the complexity should be overall reasonable.

We further apply three optimizations to this computation:

1. We merge cases where the number of probes in a gadget is strictly greater than t since in that case the inequality is always violated.[3]
2. Since cryptographic circuits often contain repeated sub-circuits, we avoid re-computation when we encounter structurally equivalent sub-circuits[4].
3. When J can be partitioned in unrelated subsets $J_1, \ldots, J_\kappa$ (J and J' are unrelated if for any $j \in J$ and $j' \in J'$, $(\{j\} \cup \mathsf{OG}_j) \cap (\{j'\} \cup \mathsf{OG}_{j'}) = \emptyset$), then we use the equality $\Pr\left[\bigwedge_{j\in J} \Delta_j\right] = \prod_{i=1}^{\kappa} \Pr\left[\bigwedge_{j\in J_i} \Delta_j\right]$ (which greatly broadens the applicability of the previous optimization).

[3] This amounts to replace in $R_{J'}$ all the elements where one of the γ_j is strictly greater than t with a Large symbol, and set the probability of this event to be the probability that any $|\mathcal{G}_j|$ exceeds t.

[4] We de-duplicate computation when there is a bijection between J' and J'' that preserves the number of gates in the gadgets and the connections between the gadgets (which can be identified by hash-consing the relevant structural information).

Finally, the complexity of computing T_i is $\mathcal{O}(k^i)$, which can make computing even B_3 very costly if k is large (*e.g.* $\sim 10^4$). Therefore, we introduce new formulas to compute T_i for $i \in \{1,2,3\}$ in Lemma 4 whose proof is given in Appendix A.

Lemma 4. *Let C be a circuit composed of gadgets $G_1, G_2, \ldots, G_k$, each satisfying the t-SNI gate-leakage property. Let $\mathcal{G}$ be a random variable representing a tuple of leaking gates sampled from* $\mathsf{LeakingGates}(C,p)$ *for some $p \in [0,1]$. We have*

$$T_1 = \sum_{i=1}^{k} \Pr[\Delta_i]$$

$$T_2 = \sum_{J \subset [\![k]\!], |J|=2} \prod_{j \in J} \Pr[\Delta_j] + \sum_{J \in \mathcal{C}_2} \left(\Pr[\Delta_J] - \prod_{j \in J} \Pr[\Delta_j] \right).$$

$$\begin{aligned} T_3 = & \sum_{J \subset [\![k]\!], |J|=3} \prod_{j \in J} \Pr[\Delta_j] \\ & + \sum_{J \in \mathcal{C}_2} \left(\left(\Pr[\Delta_J] - \prod_{j \in J} \Pr[\Delta_j] \right) \left(\sum_{i \in [\![k]\!]} \Pr[\Delta_i] - \sum_{i \in J} \Pr[\Delta_i] \right) \right) \\ & + \sum_{\substack{J=\{i,i',i''\}: \\ \neg\mathcal{U}(i,i') \wedge \neg\mathcal{U}(i,i'')}} \begin{pmatrix} \Pr[\Delta_{\{i,i',i''\}}] - \Pr[\Delta_{\{i,i'\}}] \Pr[\Delta_{i''}] - \Pr[\Delta_{\{i,i''\}}] \Pr[\Delta_{i'}] \\ -\delta_{\neg\mathcal{U}(i',i'')} \Pr[\Delta_{\{i',i''\}}] \Pr[\Delta_i] + (2 + \delta_{\neg\mathcal{U}(i',i'')}) \prod_{j \in J} \Pr[\Delta_j] \end{pmatrix} \end{aligned}$$

where $\Delta_J \equiv \bigwedge_{j \in J} \Delta_j$, $\mathcal{U}(i,i') \equiv (\{i\} \cup \mathsf{OG}_i) \cap (\{i'\} \cup \mathsf{OG}_{i'}) = \emptyset$, $\mathcal{C}_2 = \{i, i' : i, i' \in [\![k]\!], i \neq i' \wedge \mathcal{U}(i,i')\}$, and $\delta_{\neg\mathcal{U}(i,i')} = 1$ if $\neg\mathcal{U}(i,i')$, and $\delta_{\neg\mathcal{U}(i,i')} = 0$ otherwise.

Remark 1. Another approach to estimate α is to apply the Monte Carlo method: sample many tuples from $\mathsf{LeakingGates}$, evaluate whether each sampled tuple is in $\mathcal{S}_G$, and then apply the formulas of Section 3 to bound α. The drawback, as already noted, is that when α is very small, accurate estimation requires an impractically large number of samples. Monte Carlo and inclusion–exclusion are thus complementary: the latter gives tight bounds when the terms T_i for $i > s$ are small, but becomes looser as they grow, *i.e.* when the probability for a tuple to be outside $\mathcal{S}_G$ is higher. In our context, we specifically aim for small values of α, which correspond to higher security and more efficient verification. Hence the inclusion–exclusion method is generally the most suitable.

4.4 Efficient Sampling (SampleOutSG)

We now present our contribution for efficiently and uniformly sampling tuples in the set $\mathcal{T}_C \backslash \mathcal{S}_G$, that is our algorithm for $\mathsf{SampleOutSG}$.

Naive Approach. A straightforward yet inefficient implementation for the routine $\mathsf{SampleOutSG}$ is rejection sampling: sample randomly leakage tuples from $\mathsf{LeakingGates}(C, p)$ and immediately discarding those that belong to $\mathcal{S}_G$. This method can be effective when the cost of sampling is sufficiently low compared to the cost of verifying simulatability. That is, the total cost of verifying the tuples still exceeds the one of $\mathsf{SampleOutSG}$ if $T_{\text{ver}} > T_{\text{samp}}/\alpha$ where T_{ver} is the cost of verifying a tuples, while T_{samp} is the cost of sampling from $\mathsf{LeakingGates}$ and then computing α. However, the core idea of our method is to gain computation time when α is small, hence having a computation time in $\mathcal{O}(1/\alpha)$ is a significant limitation.

Improved Method. We therefore propose an improved algorithm for sampling directly in $\mathcal{T}_C \backslash \mathcal{S}_G$, whose complexity does not grow when α becomes small. Similarly to Section 3.2, we first determine the number of leaking gates in each gadget, *i.e.* the tuple $(\gamma_1, \ldots, \gamma_k) \in \mathbb{N}^k$, and then select the actual leaking gates by uniformly choosing γ_i distinct probes within each gadget G_i. This latter step is straightforward, hence we focus on sampling the random variables $\mathit{\Gamma}_i$ that represent the number of leaking gates in a gadget G_i. Without conditioning on $\mathcal{T}_C \backslash \mathcal{S}_G$, these variables are jointly independent and each $\mathit{\Gamma}_i$ follows a known binomial distribution. Since the event $(\mathcal{G} \notin \mathcal{S}_G) \equiv \bigvee_{i=1}^{k} \mathit{\Delta}_i$, the key idea is to enforce $\mathit{\Delta}_i = \top$ for at least one index $i \in [\![k]\!]$. Concretely, we first select such an i, then sample the number of leaking gates $\mathit{\Gamma}_j$ for all $j \in (\{i\} \cup \mathsf{OG}_i)$, conditioned on $\mathit{\Delta}_i$. The remaining $(\mathit{\Gamma}_\ell)_{\ell \notin (\{i\} \cup \mathsf{OG}_i)}$ can then be drawn independently from their respective binomial distributions. This strategy, however, introduces a bias: by forcing one $\mathit{\Delta}_i$ to hold (*i.e.* one SNI constraint to be violated) and independently sampling the other $\mathit{\Gamma}_j$, it increases the likelihood that multiple $\mathit{\Delta}_i$'s are simultaneously satisfied, compared to the target distribution. To correct this bias, we introduce a rejection sampling step. Since the rejection probability is small, the overhead is insignificant and the algorithm remains efficient.

Having outlined the main idea of our improved approach, we now turn to its detailed procedure.

Let $\gamma = (\gamma_1, \ldots, \gamma_k) \in \mathbb{N}^k$ represent a vector of leakage counts across the gadgets of the circuit. For each gadget G_i, recall that OG_i denotes the set of indices corresponding to the gadgets that consume its outputs. We then define the total number of SNI constraints violations as:

$$s(\gamma) = \sum_{i=1}^{k} \delta_i(\gamma)$$

where $\delta_i(\gamma) = 1$ if $\gamma_i + \sum_{\ell \in \mathsf{OG}_i} \gamma_\ell > t$, and 0 otherwise. Hence, $s(\gamma) = 0$ corresponds to a fully simulatable configuration under the SNI assumptions, while $s(\gamma) > 0$ denotes an SNI violation.

Our goal is to construct a randomized algorithm that samples leakage vectors γ according to the conditional distribution:

$$\Pr[\mathit{\Gamma}' = \gamma] = \Pr[\mathit{\Gamma} = \gamma \mid \mathit{\Delta}],$$

where Γ denotes the original (unconditioned) leakage distribution across gadgets and Δ is the event that at least one violation occurs. This implies the following distribution for Γ':

$$\Pr[\Gamma' = \gamma] = \begin{cases} 0 & \text{if } s(\gamma) = 0, \\ \dfrac{\Pr[\Gamma = \gamma]}{\Pr[\Delta]} & \text{otherwise,} \end{cases} \tag{4}$$

which directly follows from Bayes' rule (since $\Pr[\Delta \mid \Gamma = \gamma] = 1$).

To sample from the distribution $\Pr[\Gamma' = \gamma]$, we leverage the fact that for each event Δ_i (*i.e.* a SNI constraint is violated for gadget G_i), we can sample from the conditional distribution $\Pr[\Gamma = \gamma \mid \Delta_i]$. The key idea is to sample an index i to select an event Δ_i from the probability $\Pr[\Delta_i]$, and then draw a sample from $\Pr[\Gamma = \gamma \mid \Delta_i]$. Since such samples may be over-represented, we perform a rejection step to correct the distribution. The full sampling algorithm is detailed in Algorithm 3.

Algorithm 3 Sampling for the vector of cardinals

Input: Circuit C made of gadgets $\{G_1, \ldots, G_k\}$; random probing probability $p \in [0, 1]$; SNI parameter t

Output: Sample $\Gamma' \sim \Pr[\Gamma = \gamma \mid \Delta]$

1: **repeat**
2: $\quad I \leftarrow$ Sample from $\mathsf{Categorical}\left(\frac{\Pr[\Delta_1]}{\sum_{\ell=1}^{k} \Pr[\Delta_\ell]}, \ldots, \frac{\Pr[\Delta_k]}{\sum_{\ell=1}^{k} \Pr[\Delta_\ell]}\right)$
3: $\quad \Gamma'_I \leftarrow \mathsf{SampleCards}\left((|G_1|, \ldots, |G_k|), I\right)$ ▷ where $\Pr[\Gamma'_I = \gamma] = \Pr[\Gamma = \gamma \mid \Delta_I]$
4: $\quad A_I \xleftarrow{\$} \mathcal{B}(\frac{1}{s(\Gamma')})$ ▷ where $A_I = \top$ with probability $\frac{1}{s(\Gamma')}$
5: **until** $A_I = \top$
6: **return** Γ'

The first instruction corresponds to a sampling from a categorical distribution, where the probabilities of the events Δ_i can be expressed as

$$\Pr[\Delta_i] = \Pr\left[\left(\Gamma_i + \sum_{\ell \in \mathsf{OG}_i} \Gamma_\ell\right) > t\right]$$

with $\left(\Gamma_i + \sum_{\ell \in \mathsf{OG}_i} \Gamma_\ell\right) \sim \mathcal{B}\left(|G_i| + \sum_{\ell \in \mathsf{OG}_i} |G_\ell|, p\right)$, where p denotes the leaking probability for each gate.

In the procedure $\mathsf{SampleCards}$, the idea is therefore to sample the cardinalities of the probes in G_I and in the gadgets in OG_I from their binomial laws, conditioned on the constraint that their total sum must exceed the threshold t. The cardinalities of all the remaining gadgets can then be sampled independently, each according to a binomial distribution determined by the gadget size and leakage probability p, without additional constraints.

This algorithm remains efficient as long as $\Pr[A_I = \top]$ is not too small (since the rejection probability is precisely $\Pr[A_I = \bot]$). The condition $A_I = \bot$

occurs only if $s(\Gamma') > 1$, which is somewhat unlikely under the assumption that Δ itself is a rare event (this is precisely why we rely on this algorithm). Indeed, Δ corresponds to the satisfaction of at least one inequality, whereas $s(\Gamma') > 1$ requires at least two inequalities to hold simultaneously (*i.e.* two SNI constraints violated). Although the sampling of Γ' is constrained by the event Δ_I, and the Δ_I's are not independent, the overall rejection probability should remain reasonably low.

Lemma 5 (Correctness of the Sampling Algorithm). *The output distribution Γ' of Algorithm 3 corresponds to the conditional distribution $\Gamma \mid \Delta$ where Γ is the vector of leakage counts sampled from* $\mathsf{LeakingGates}(C, p)$ *and Δ denotes the event that at least one local SNI constraint is violated.*

Proof. Let Γ' be the output of Algorithm 3. Let us first handle the particular case $s(\gamma) = 0$: since $s(\Gamma) = 0$ implies $\neg\Delta$, $\Pr[\Gamma = \gamma \mid \Delta] = 0$ in this case, and $\Pr[\Gamma' = \gamma] = 0$ since $\Pr[\Gamma'_i = \gamma] = 0$ for all i.

Next, for all other γ, it holds

$$\begin{aligned}\Pr[\Gamma' = \gamma] = \Pr[\Gamma'_I = v \mid A_I = \top] &= \frac{\sum_{i=1}^{k} \Pr[\Gamma'_i = \gamma \wedge A_i = \top] \Pr[I = i]}{\Pr[A_I = \top]} \\ &= \frac{\sum_{i=1}^{k} \Pr[A_i = \top \mid \Gamma'_i = \gamma] \Pr[\Gamma'_i = \gamma] \Pr[\Delta_i]}{\Pr[A_I = \top] \sum_{i=1}^{k} \Pr[\Delta_i]} \\ &= \frac{1/s(\gamma) \sum_{i=1}^{k} \Pr[\Gamma = \gamma \mid \Delta_i] \Pr[\Delta_i]}{\Pr[A_I = \top] \sum_{i=1}^{k} \Pr[\Delta_i]}\end{aligned}$$

where the second line expands the distribution of I and the last equality comes from the sampling of A_i. Next, applying Bayes' rule and using the definitions of $\delta_i(v)$, $s(v)$, we get:

$$\begin{aligned}\Pr[\Gamma' = \gamma] &= \frac{1/s(\gamma) \sum_{i=1}^{k} \Pr[\Delta_i \mid \Gamma = \gamma] \Pr[\Gamma = \gamma]}{\Pr[A_I = \top] \sum_{i=1}^{k} \Pr[\Delta_i]} \\ &= \frac{\left(\sum_{i=1}^{k} \delta_i(\gamma)/s(\gamma)\right) \Pr[\Gamma = \gamma]}{\Pr[A_I = \top] \sum_{i=1}^{k} \Pr[\Delta_i]} = \frac{\Pr[\Gamma = \gamma]}{\Pr[A_I = \top] \sum_{i=1}^{k} \Pr[\Delta_i]}.\end{aligned}$$

which concludes the proof that the algorithm is correct: if $s(\gamma) \neq 0$, we satisfy Equation 4, since $\Pr[\Gamma' = \gamma] = \eta \Pr[\Gamma = \gamma]$ for some η. □

5 Experiments and Comparisons

We developed perseus, a verification tool that estimates the random probing security of masked AES-128 circuits (code available in the Supplementary Material). Our AES design follows the construction of Rivain and Prouff [36] using

an abritrary number of shares n, with a few modifications. Most importantly, we added refresh gadgets between linear operations and between each pair of successive multiplications in the S-box, where the output of one multiplication serves as input to the next. Further, refresh gadgets are implemented with the $n \log_2(n)$ (SNI) refresh gadget of [8,32], and multiplications are instantiated with the ISW gadget [28] or the variants presented in Section 5.2. We study round-reduced variants as well as circuits without key schedule. The algorithms of the gadgets used in the AES circuits are provided in Appendix B.

perseus can be configured through several parameters. The user may select one of the three backends described in Section 3.1, specify the leakage rate p and choose between the wire or gate leakage model. The tool relies on Monte Carlo sampling, with a tunable number of samples for the main estimation and, when relevant, an additional sampling phase to estimate α. The confidence level of the bound is set through a parameter δ, while the inclusion–exclusion expansion, when relevant, can be limited to depth one or three. Finally, the tool supports parallel execution, with a configurable degree of parallelism.

In the following, we report on experiments comparing the three available backends, as well as the security and complexity achieved with the different multiplication gadgets. We then evaluate our full AES-128 implementation against state-of-the-art compiler-generated implementations, both in terms of security and efficiency. All experiments were conducted on a workstation equipped with an AMD Ryzen Threadripper PRO 7995WX CPU (96 cores, 192 threads) and 512 GB of RAM.

For these experiments, we set most parameters (circuit design, number of shares, and noise rate) to target roughly 32-bit security. This choice reflects the fact that side-channel attacks typically amplify the advantage of single-trace distinguishers through repeated executions. However, the number of exploitable executions is practically limited, and in real scenarios most often lies in the range 2^8 to 2^{24} [17,20,34].

5.1 Comparison of perseus' Backends and round-reduced AES

We begin by comparing the performance of perseus across its three integrated backends: maskVerif, verifMSI, and favom. To this end, we evaluate AES-128 circuits with different numbers of rounds, both with and without the key schedule (denoted KS in Table 1), using the ISW multiplication [28]. Table 1 reports, for each configuration, the number of non-random gates, the upper bound on random probing security, the tightness achieved by the Monte Carlo method, and the verification time (using 192 threads/processes, except for verifMSI, which does not support parallelization). Entries marked with "–" indicate that the computation did not complete within one hour.

The data show that favom consistently achieves the fastest runtimes, while the reported security bounds remain very similar across all settings. This demonstrates that the backends have similar false-positive rates. For all but the single-round cases, the verifMSI backend did not complete the circuit construction step in one hour, hence no sample could be verified.

Table 1. Comparison of the three backends integrated into perseus for the random probing security evaluation of AES-128 implementations with ISW multiplication [28], under the gate leakage model with leakage rate $p = 2^{-12}$, confidence parameter $\delta = 10^{-3}$, $n = 6$ shares, 2^{16} samples.

Circuit	Backend	Security bound ε^U	Tightness $\log_2(\frac{\varepsilon^U}{\varepsilon^L})$	Timing
1-round AES-128 ➤ 11 168 gates	maskVerif	$2^{-38.2}$	0.7	10s
	verifMSI	$2^{-38.5}$	0.8	44min
	favom	$2^{-38.3}$	0.7	2s
4-round AES-128 ➤ 43 712 gates	maskVerif	$2^{-36.4}$	0.7	49s
	verifMSI	–	–	–
	favom	$2^{-36.4}$	0.8	5s
full AES-128 ➤ 106 752 gates	maskVerif	$2^{-35.0}$	0.7	2min
	verifMSI	–	–	–
	favom	$2^{-35.0}$	0.7	11s
full AES-128 + KS ➤ 138 640 gates	maskVerif	$2^{-34.6}$	0.7	3min
	verifMSI	–	–	–
	favom	$2^{-34.9}$	0.8	17s

Moreover, the results suggest that the random probing security advantage increases linearly with the number of rounds, which is expected when the circuit does not suffer from composition issues.

5.2 Comparison between Different Multiplications

We next use perseus, with its favom backend, to evaluate the random probing security of an AES-128 implementation (including the key schedule) built with different multiplication gadgets. Our comparison spans several dimensions: the security upper bound, the probability α, the tightness achieved for a given number of samples, the verification time, and the circuit complexity in terms of random and non-random gates. We consider five implementations: (i) an AES-128 design based on the classical ISW multiplication [28] (recalled in Algorithm 12), and (ii–v) four variants inspired by the proposal of [8], further optimized in [21], which we collectively denote RPMult (and that is recalled in Algorithm 14). Each RPMult variant is paired with a different refresh strategy: (ii) the simple gadget of [36] (simple), using $n - 1$ random values; (iii) the refresh gadget from [8,32] (log) with a randomness complexity of $\Theta(n \log_2 n)$ random values; (iv) the half refresh, which applies randomness to successive pairs of shares, with a randomness complexity of $\lceil \frac{n}{2} \rceil$; (v) the circular refresh gadget of [7] (circular), using n random values (reduced to a single value when $n = 2$). The algorithm for each refresh gadget is provided in Appendix B.

The table reports results for a leakage rate $p = 2^{-10}$, computed for numbers of shares $n \in \{4, 6, 8\}$, and also indicates for each n the maximum p at which

Table 2. Comparison of perseus results on the random probing security evaluation of AES-128 implementations with different multiplication gadgets, under the gate leakage model with confidence parameter $\delta = 10^{-3}$.

perseus inputs		**perseus outputs**				**Complexity**	
Circuit	Samples	Sec. bound ε^U	α^U	Tightness $\log_2(\frac{\varepsilon^U}{\varepsilon^L})$	Timing	Rand.	Gates
$n = 4$ shares, $p = 2^{-10}$							
ISW	2^{16}	$2^{-14.3}$	$2^{-9.4}$	0.3	9s	17 904	68 872
RPMult - all	2^{16}	$2^{-16.0}$	$2^{-7.9}$	0.7	10s	24 304	81 672
$n = 4$ shares, $p = 2^{-14}$							
ISW	2^{16}	$2^{-30.3}$	$2^{-25.3}$	0.3	7s	17 904	68 872
RPMult - all	2^{16}	$2^{-32.1}$	$2^{-23.8}$	0.8	8s	24 304	81 672
$n = 6$ shares, $p = 2^{-10}$							
ISW	2^{16}	$2^{-22.8}$	$2^{-14.6}$	0.7	19s	34 932	138 640
RPMult - half	2^{25}	$2^{-29.1}$	$2^{-10.7}$	1.1	133min	60 532	189 840
RPMult - simple	2^{25}	$2^{-28.9}$	$2^{-10.7}$	1.0	144min	60 532	189 840
RPMult - log	2^{25}	$2^{-29.5}$	$2^{-10.7}$	1.3	129min	60 532	189 840
RPMult - circular	2^{25}	$2^{-28.9}$	$2^{-10.0}$	1.5	149min	66 932	202 640
$n = 6$ shares, $p = 2^{-12}$							
ISW	2^{16}	$2^{-34.9}$	$2^{-26.4}$	0.8	17s	34 932	138 640
RPMult - half	2^{25}	$2^{-41.1}$	$2^{-22.4}$	1.3	115min	60 532	189 840
RPMult - simple	2^{25}	$2^{-40.9}$	$2^{-22.4}$	1.2	100min	60 532	189 840
RPMult - log	2^{25}	$2^{-41.0}$	$2^{-22.4}$	1.2	105min	60 532	189 840
RPMult - circular	2^{25}	$2^{-40.8}$	$2^{-21.6}$	1.6	109min	66 932	202 640
$n = 8$ shares, $p = 2^{-10}$							
ISW	2^{19}	$2^{-30.5}$	$2^{-18.4}$	1.0	1min	61 712	240 712
RPMult - half	2^{23}	$2^{-34.2}$	$2^{-14.0}$	$+\infty$	82min	100 112	317 512
RPMult - simple	2^{23}	$2^{-33.6}$	$2^{-13.4}$	$+\infty$	91min	106 512	330 312
RPMult - log	2^{23}	$2^{-33.0}$	$2^{-12.8}$	$+\infty$	90min	112 912	343 112
RPMult - circular	2^{23}	$2^{-33.0}$	$2^{-12.8}$	$+\infty$	92min	112 912	343 112

a security level of at least 30 bits is reached. As expected, the RPMult variants involve a larger number of gates than the classical ISW multiplication. While they provide higher random probing security levels, this comes at the cost of requiring more samples to reach a reasonable tightness. In all cases, we progressively increased the number of samples following a geometric progression, until the tightness approached 1 and subject to a runtime limit of three hours. For higher complexities (notably at $n = 8$ shares), no failures could be observed for leakage rate 2^{-10} in the RPMult variants, which explains the reported tightness values of $+\infty$. In such cases, the ratio ε^U/α^U is inversely proportional to the number of samples (as discussed in Section 3.2). Therefore, a tighter security

bound can be achieved by increasing the number of samples (*i.e.* the verification time) or reducing α. For the latter, it is interesting to note that smaller gadgets lead to smaller α (for fixed n and p): even if RPMult gadgets are more secure than ISW, this security does not appear in the t-SNI characterization we rely on when computing α, while the larger size leads to larger probe tuples and therefore increases the probability of exceeding the threshold t.

Among the different refresh strategies, RPMult combined with the half refresh appears to offer the best trade-off between security and complexity. Indeed, it has the lowest complexity overhead and achieves security levels similar to the other RPMult refreshing strategies.

5.3 Comparison with State-Of-The-Art Methods

Having identified in the previous experiments the multiplication gadget RPMult combined with the half refresh gadget as an interesting trade-off, we now compare its performance in terms of both security and complexity with the most efficient state-of-the-art random probing secure compilers.

The expanding compiler of [11,14,15] relies on simulation-based properties named threshold random probing composability and random probing expandability, which allow one to achieve arbitrarily high security levels (for a fixed leakage rate). However, the resulting implementations come with prohibitive complexity. In the same vein, the general RPC framework based on PDTs yields tighter bounds but does not scale for large circuits (the verification complexity is exponential in the number of parallel wires). With a different strategy, Berti et al. [18] use leakage diagrams to directly evaluate the random probing security of an implementation based on a variant of the ISW multiplication. Yet, even with a leakage rate of $p = 2^{-15}$, the smallest possible advantage $\varepsilon = 2^{-31.0} \cdot |C|$ (where $|C|$ denotes the number of gates of the target circuit C) is only reached for $n = 23$ shares, limiting the practical relevance. As for the recent tool INDIANA [9], it does not scale to large circuits: its use for verifying an AES-128 circuit is limited to two shares, and relies on the *cycle accurate* random probing model, which is weaker than the probing model (it constraints all probes in a tuple to belong to a small subset of the circuit).

For these reasons, our experiments focus instead on two compilers that we identify as state-of-the-art: JMB24 [29] and BNR25 [16]. JMB24 builds upon a variant of the multiplication from [8] and upper bounds the mutual information between the leakage and a single byte of the secret key. BNR25 also builds upon a variant of the multiplication from [8] but relies on the simulation-based notion of *cardinal random probing composability*, combining it with carefully chosen permutations of sharings; from this, the compiler computes threshold random probing composable security, which in turn serves as an upper bound to the random probing security.

Table 3 compares the three frameworks in terms of implementations and computed security bounds, for two leakage rates $p \in \{2^{-10}, 2^{-15}\}$. In each case, the number of shares is selected so that the random probing security guarantees an advantage below 2^{-32}, with $\delta = 10^{-3}$. Since JMB24 and BNR25 evaluate

their security level in the wire random probing model, our comparison is also carried out in this setting for consistency.

The experiments with JMB24 are straightforward, as the random probing security level directly follows from an analytical formula depending on the number of shares n and the leakage rate p. For BNR25, we relied on the authors' scripts[5] to compute the best achievable complexity for the targeted security level. We emphasize that our comparison is somewhat favorable to BNR25, since we did not account for the additional randomness required for the permutations (which is not secret but still impacts the overall cost).

Table 3. Comparison of three state-of-the-art frameworks in terms of complexity and computed random probing security bounds for masked AES-128 (without key schedule), with $\delta = 10^{-3}$.

	Tools' outputs				**Complexity**		
	Timing (samples)	Sec. bound (ε^U)	α^U	Tightness ($\log_2(\frac{\varepsilon^U}{\varepsilon^L})$)	Shares	Rand.	Gates
$p = 2^{-10}$, $\lambda = 32$							
JMB24 [29]	< 0.1s	$2^{-32.2}$	N/A	N/A	15	509 495	1 892 005
BNR25 [16]	29s	$2^{-33.9}$	N/A	N/A	10	200 928	846 880
perseus (wire)	153min (2^{25})	$2^{-32.2}$	$2^{-10.0}$	$+\infty$	7	59 776	193 024
perseus (gate)	135min (2^{25})	$2^{-34.0}$	$2^{-12.7}$	6.1	7	57 216	187 904
$p = 2^{-15}$, $\lambda = 32$							
JMB24 [29]	< 0.1s	$2^{-32.0}$	N/A	N/A	9	152 156	605 524
BNR25 [16]	3s	$2^{-34.1}$	N/A	N/A	6	127 680	440 800
perseus (wire)	8s (2^{19})	$2^{-38.1}$	$2^{-26.6}$	0.9	4	18 176	62 976
perseus (gate)	5s (2^{16})	$2^{-36.3}$	$2^{-28.1}$	0.8	4	18 176	62 976

For both leakage models, perseus achieves 32-bit security with our implementation based on SNI gadgets, using the RPMult multiplication with our half refresh. Concretely, this requires only 4 shares for a leakage rate of 2^{-15} (in both gate and wire leakage models) and 7 shares for a leakage rate of 2^{-10}. This represents a significant improvement over previous techniques, which require a larger number of shares and therefore higher complexity to achieve the same provable security level. It is worth noting that the number of samples used in our experiments with perseus varied across settings. In practice, we progressively increased the number of samples until achieving 32 bits of security with a target tightness of 1, while keeping the runtime below three hours.

Besides the comparison to other tools, an important observation is that the security bounds in the gate and in the wire leakage model are very close. This highlights the practical interest of working directly in the gate leakage model: without incurring any cost, it avoids the loss (squaring p) of reducing gate leakage

[5] `https://github.com/CryptoExperts/AC25-cardinal-rp-compiler`

security to wire leakage security. Further, the verification time with perseus is significantly lower in the gate leakage model than in the wire leakage model, since the number of gates is smaller than the number of wires, which reduces α.

6 Conclusion

In this work, we introduced a new methodology for evaluating random probing security that achieves significantly tighter bounds than the state of the art, while keeping the computational cost low. Our approach scales to complete masked AES implementations for high numbers of shares, demonstrating that practical evaluations at this level of complexity are feasible. We can also evaluate the impact of changes to the masking scheme, such as the choice of refresh or multiplication gadgets. Moreover, we showed that moving from the wire leakage model to the gate leakage model has little impact on the reported security levels, yet provides a much tighter reduction to the noisy leakage model.

Our methodology proceeds in two steps. First, we derive a random probing security bound from the composition of SNI gadgets, analogous to the reduction of random probing security to region probing security. This yields an upper bound α^U, which can become very small with enough shares and low leakage rates, but is generally not tight. Second, we refine this bound into ε^U using a Monte Carlo factor: by sampling, we tighten the security bound ε^U, by a refinement factor that scales with computation time. In practice, this factor (ε^U/α^U) can be reduced to about 2^{-22} within three hours. For instance, with $n = 6$ shares and leakage rate $p = 2^{-12}$, our method yields a tight security bound $\varepsilon^U \approx 2^{-40}$. At higher leakage rates, however, α^U is less tight; and for circuits with high security levels (*e.g.* RPMult with $n = 7$ and $p = 2^{-10}$), the refinement saturates, producing looser security bounds (*e.g.* $2^{-40} \leq \epsilon \leq 2^{-34}$). This limitation reflects a broader open question: what security levels should we target in practice for side-channel resistance, especially when accounting for the measurement cost of real attacks?

Finally, our findings open promising directions for future research. First, refining the pruning technique, for instance by exploiting finer-grained composition strategies or smaller gadgets, could further improve tightness without increasing verification complexity. Second, optimized designs for masked circuits could be explored, by minimizing the use of refresh gadgets while preserving high security levels. Third, developing a compositional reasoning for random probing security (rather than relying on random probing composability) would allow us to verify even larger masked circuits, such as circuits encrypting long messages.

Acknowledgements This work has been funded in part by the European Union through the ERC project 101077506 (acronym AMAskZONE). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Ajtai, M.: Secure computation with information leaking to an adversary. In: Fortnow, L., Vadhan, S.P. (eds.) 43rd ACM STOC. pp. 715–724. ACM Press (Jun 2011). https://doi.org/10.1145/1993636.1993731
2. Ananth, P., Ishai, Y., Sahai, A.: Private circuits: A modular approach. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 427–455. Springer, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96878-0_15
3. Andrychowicz, M., Dziembowski, S., Faust, S.: Circuit compilers with $O(1/\log(n))$ leakage rate. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 586–615. Springer, Berlin, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49896-5_21
4. Barthe, G., Belaïd, S., Cassiers, G., Fouque, P.A., Grégoire, B., Standaert, F.X.: maskVerif: Automated verification of higher-order masking in presence of physical defaults. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019, Part I. LNCS, vol. 11735, pp. 300–318. Springer, Cham (Sep 2019). https://doi.org/10.1007/978-3-030-29959-0_15
5. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y.: Verified proofs of higher-order masking. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 457–485. Springer, Berlin, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_18
6. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 116–129. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978427
7. Barthe, G., Dupressoir, F., Faust, S., Grégoire, B., Standaert, F.X., Strub, P.Y.: Parallel implementations of masking schemes and the bounded moment leakage model. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part I. LNCS, vol. 10210, pp. 535–566. Springer, Cham (Apr / May 2017). https://doi.org/10.1007/978-3-319-56620-7_19
8. Battistello, A., Coron, J.S., Prouff, E., Zeitoun, R.: Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 23–39. Springer, Berlin, Heidelberg (Aug 2016). https://doi.org/10.1007/978-3-662-53140-2_2
9. Beierle, C., Feldtkeller, J., Guinet, A., Güneysu, T., Leander, G., Richter-Brockmann, J., Sasdrich, P.: INDIANA - verifying (random) probing security through indistinguishability analysis. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part VIII. LNCS, vol. 15608, pp. 33–63. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91101-9_2
10. Belaïd, S., Cassiers, G., Mutschler, C., Rivain, M., Roche, T., Standaert, F.X., Taleb, A.R.: SoK: A methodology to achieve provable side-channel security in real-world implementations. CiC **2**(1), 4 (2025). https://doi.org/10.62056/aebngy4e-
11. Belaïd, S., Coron, J.S., Prouff, E., Rivain, M., Taleb, A.R.: Random probing security: Verification, composition, expansion and new constructions. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 339–368. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2_12
12. Belaïd, S., Mercadier, D., Rivain, M., Taleb, A.R.: IronMask: Versatile verification of masking security. In: 2022 IEEE Symposium on Security and Privacy. pp.

142–160. IEEE Computer Society Press (May 2022). https://doi.org/10.1109/SP46214.2022.9833600
13. Belaïd, S., Rivain, M., Rossi, M.: New techniques for random probing security and application to raccoon signature scheme. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part VIII. LNCS, vol. 15608, pp. 94–123. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91101-9_4
14. Belaïd, S., Rivain, M., Taleb, A.R.: On the power of expansion: More efficient constructions in the random probing model. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part II. LNCS, vol. 12697, pp. 313–343. Springer, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77886-6_11
15. Belaïd, S., Rivain, M., Taleb, A.R., Vergnaud, D.: Dynamic random probing expansion with quasi linear asymptotic complexity. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part II. LNCS, vol. 13091, pp. 157–188. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92075-3_6
16. Belaïd, S., Normand, V., Rivain, M.: Masked circuit compiler in the cardinal random probing composability framework. Cryptology ePrint Archive, Paper 2025/1747 (2025), https://eprint.iacr.org/2025/1747
17. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. J. Cryptogr. Eng. **10**(2), 163–188 (2020)
18. Berti, F., Faust, S., Orlt, M.: Provable secure parallel gadgets. IACR TCHES **2023**(4), 420–459 (2023). https://doi.org/10.46586/tches.v2023.i4.420-459
19. Cassiers, G., Faust, S., Orlt, M., Standaert, F.X.: Towards tight random probing security. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 185–214. Springer, Cham, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84252-9_7
20. Cassiers, G., Momin, C.: The SMAesH dataset. Cryptology ePrint Archive, Report 2024/1521 (2024), https://eprint.iacr.org/2024/1521
21. Cassiers, G., Standaert, F.X.: Towards globally optimized masking: From low randomness to low noise rate. IACR TCHES **2019**(2), 162–198 (2019). https://doi.org/10.13154/tches.v2019.i2.162-198, https://tches.iacr.org/index.php/TCHES/article/view/7389
22. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 398–412. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_26
23. Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: From probing attacks to noisy leakage. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 423–440. Springer, Berlin, Heidelberg (May 2014). https://doi.org/10.1007/978-3-642-55220-5_24
24. Dziembowski, S., Faust, S., Zebrowski, K.: Simple refreshing in the noisy leakage model. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part III. LNCS, vol. 11923, pp. 315–344. Springer, Cham (Dec 2019). https://doi.org/10.1007/978-3-030-34618-8_11
25. Faust, S., Grosso, V., Merino Del Pozo, S., Paglialonga, C., Standaert, F.X.: Composable masking schemes in the presence of physical defaults & the robust probing model. IACR TCHES **2018**(3), 89–120 (2018). https://doi.org/10.13154/tches.v2018.i3.89-120, https://tches.iacr.org/index.php/TCHES/article/view/7270

26. Goubin, L., Patarin, J.: DES and differential power analysis (the "duplication" method). In: Koç, Çetin Kaya., Paar, C. (eds.) CHES'99. LNCS, vol. 1717, pp. 158–172. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48059-5_15
27. Goudarzi, D., Joux, A., Rivain, M.: How to securely compute with noisy leakage in quasilinear complexity. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part II. LNCS, vol. 11273, pp. 547–574. Springer, Cham (Dec 2018). https://doi.org/10.1007/978-3-030-03329-3_19
28. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Berlin, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4_27
29. Jahandideh, V., Mennink, B., Batina, L.: An algebraic approach for evaluating random probing security with application to AES. IACR TCHES **2024**(4), 657–689 (2024). https://doi.org/10.46586/tches.v2024.i4.657-689
30. Knichel, D., Sasdrich, P., Moradi, A.: SILVER - statistical independence and leakage verification. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part I. LNCS, vol. 12491, pp. 787–816. Springer, Cham (Dec 2020). https://doi.org/10.1007/978-3-030-64837-4_26
31. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 388–397. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_25
32. Mathieu-Mahias, A.: Sécurisation des implémentations d'algorithmes cryptographiques pour les systèmes embarqués. Ph.D. thesis, Université Paris-Saclay, France (2021), https://lmv.math.cnrs.fr/soutenance-de-these-daxel-mathieu-mahias/, soutenance : 6 décembre 2021
33. Meunier, Q.L., Taleb, A.R.: Verifmsi: Practical verification of hardware and software masking schemes implementations. In: di Vimercati, S.D.C., Samarati, P. (eds.) Proceedings of the 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, Italy, July 10-12, 2023. pp. 520–527. SCITEPRESS (2023). https://doi.org/10.5220/0012138600003555, https://doi.org/10.5220/0012138600003555
34. Picek, S., Perin, G., Mariot, L., Wu, L., Batina, L.: Sok: Deep learning-based physical side-channel analysis. ACM Comput. Surv. **55**(11), 227:1–227:35 (2023)
35. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 142–159. Springer, Berlin, Heidelberg (May 2013). https://doi.org/10.1007/978-3-642-38348-9_9
36. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Berlin, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-15031-9_28
37. Scholz, F.: Confidence bounds & intervals for parameters relating to the binomial, negative binomial, poisson and hypergeometric distributions with applications to rare events (2008)

A Proof of Lemma 4

Proof (Proof of Lemma 4). As a starting point, we write down the expression for T_1 from Equation 3, whose computational cost is proportional to k: $T_1 = \sum_{i=1}^{k} \Pr[\Delta_i]$.

Then, we consider T_2, which gives a double summation:

$$T_2 = \sum_{i=1}^{k} \sum_{i'=i+1}^{k} \Pr[\Delta_i \wedge \Delta_{i'}].$$

In order to avoid the quadratic cost in k, we draw on the previous optimization (3) for computing $\Pr[\Delta_j \wedge \Delta_{j'}]$: if G_j and $G_{j'}$ are unrelated, then $\Pr[\Delta_j \wedge \Delta_{j'}] = \Pr[\Delta_j]\Pr[\Delta_{j'}]$. With $\mathcal{U}(j,j') \equiv (\{j\} \cup \mathsf{OG}_j) \cap (\{j'\} \cup \mathsf{OG}_{j'}) = \emptyset$ and $\mathcal{C}_2 = \{j, j' : j, j' \in [\![k]\!], j \neq j' \wedge \neg\mathcal{U}(j,j')\}$, we obtain

$$\begin{aligned} T_2 &= \sum_{\substack{J \subset [\![k]\!], |J|=2 \\ J \notin \mathcal{C}_2}} \prod_{j \in J} \Pr[\Delta_j] + \sum_{J \in \mathcal{C}_2} \Pr[\Delta_J] \\ &= \sum_{J \subset [\![k]\!], |J|=2} \prod_{j \in J} \Pr[\Delta_j] + \sum_{J \in \mathcal{C}_2} \left(\Pr[\Delta_J] - \prod_{j \in J} \Pr[\Delta_j] \right). \end{aligned}$$

Since the size of $\mathcal{C}_2$ is proportional to k (assuming a constant bound on the fan-out of gadgets), this can be computed in time proportional to k (for the first sum, one may compute it in time $\mathcal{O}(k)$ using, *e.g.* dynamic programming).

For T_3, we can use a similar technique. Given some $J \subset [\![k]\!]$ with $|J| = 3$, we distinguish three cases. In the first case, all elements of J are pairwise unrelated, that is for any distinct $i, i' \in J$, $\mathcal{U}(i,i')$, hence $\Pr[\Delta_J] = \prod_{j \in J} \Pr[\Delta_j]$. In the second case, $J = \{i\} \cup J'$ with $J' \in \mathcal{C}_2$ and $\mathcal{U}(i,i')$ for all $i' \in J'$, hence $\Pr[\Delta_J] = \Pr[\Delta_i]\Pr[\Delta_{J'}]$. Finally, in the third case, $J = \{i, i', i''\}$ with $\neg\mathcal{U}(i,i') \wedge \neg\mathcal{U}(i',i'')$, in which case we cannot factorize the computation of $\Pr[\Delta_J]$. We can therefore split the computation of T_3 in the following terms:

$$T_3 = \sum_{\substack{J [\![k]\!], |J|=3: \\ \forall i,i' \in J, \mathcal{U}(i,i')}} \prod_{j \in J} \Pr[\Delta_j] + \sum_{J \in \mathcal{C}_2} \sum_{\substack{i \in [\![k]\!] \setminus J: \\ \forall j \in J, \mathcal{U}(i,j)}} \Pr[\Delta_J]\Pr[\Delta_i] + \sum_{\substack{J=\{i,i',i''\} \subset [\![k]\!]: \\ \neg\mathcal{U}(i,i') \wedge \neg\mathcal{U}(i,i'')}} \Pr[\Delta_J] \tag{5}$$

$$= \sum_{J \subset [\![k]\!], |J|=3} \prod_{j \in J} \Pr[\Delta_j] \tag{6}$$

$$+ \sum_{J \in \mathcal{C}_2} \left(\left(\Pr[\Delta_J] - \prod_{j \in J} \Pr[\Delta_j] \right) \left(\sum_{i \in [\![k]\!]} \Pr[\Delta_i] - \sum_{i \in J} \Pr[\Delta_i] \right) \right) \tag{7}$$

$$+ \sum_{\substack{J=\{i,i',i''\} \subset [\![k]\!]: \\ \neg\mathcal{U}(i,i') \wedge \neg\mathcal{U}(i,i'')}} \begin{pmatrix} \Pr[\Delta_{\{i,i',i''\}}] - \Pr[\Delta_{\{i,i'\}}]\Pr[\Delta_{i''}] - \Pr[\Delta_{\{i,i''\}}]\Pr[\Delta_{i'}] \\ -\delta_{\neg\mathcal{U}(i',i'')} \Pr[\Delta_{\{i',i''\}}]\Pr[\Delta_i] + (2 + \delta_{\neg\mathcal{U}(i',i'')}) \prod_{j \in J} \Pr[\Delta_j] \end{pmatrix} \tag{8}$$

where $\delta_{\neg\mathcal{U}(i,i')} = 1$ if $\neg\mathcal{U}(i,i')$, and $\delta_{\neg\mathcal{U}(i,i')} = 0$ otherwise. Each of the terms in Equation 5 corresponds to one of the cases, and the cases are then re-written in optimized form in terms (6), (7) and (8). The first case is easy, but (6) includes extraneous terms for sets J which do not correspond to the first case. For the second case, we use a similar approach to the computation of T_2, and add a new gadget outside the element of $\mathcal{C}_2$ in (7), which results in the correct computation for the second case (including compensation of the first term) but adds extraneous terms for the third case. Finally, for the third case, we have to subtract all extraneous terms added previously, resulting in (8). Overall, the number of sets J for the sum of (8) is proportional to k (assuming a constant bound on the gadget fan-out), hence all terms are computed in time proportional to k. □

B Gadgets

Refresh gadgets. Algorithm 4 introduces our half refresh gadget, which uses only $\lceil n/2 \rceil$ random values. The three subsequent refresh gadgets rely on the prior generation of a zero-sharing, *i.e.* a sharing of 0, that is then added component-wise to the input sharing. The simple refresh gadget of Algorithm 5 requires $n-1$ random values, while the log refresh gadget of Algorithm 7 has a randomness complexity of $\Theta(n \log_2 n)$. Finally, the circular refresh gadget of Algorithm 9 requires n random values, except in the special case $n = 2$, where only a single random value is sufficient.

Multiplication gadgets. Algorithm 11 is the template for our two multiplications gadgets: the ISW multiplication (Algorithm 12) and RPMult (Algorithm 14), which relies on a matrix refreshing (Algorithm 13).

AES. The main components of the AES are the S-box (Algorithm 16) and MixColumn (Algorithm 18), since ShiftRows does not create any leakage by itself (it is only wiring). AddRoundKey is implemented with Xor (Algorithm 17), while the key schedule uses the S-box, Xor and Affine (for round constants).

Algorithm 4 half refresh gadget

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x$

if $n = 2$ **then**
    $r \leftarrow \$$
    return $(x_1 + r,\ x_2 - r)$
$m \leftarrow \lfloor n/2 \rfloor$
$r_1, \ldots, r_m \leftarrow \$$
for $i \leftarrow 1$ **to** m **do**
    $z_{2i-1} \leftarrow x_{2i-1} + r_i$
    $z_{2i} \leftarrow x_{2i} - r_i$
if n is odd **then**
    $r \leftarrow \$$
    $z_n \leftarrow x_n$
    $z_1 \leftarrow z_1 - x_n$
return $(z_1, \ldots, z_n)$

Algorithm 5 simple refresh gadget

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x$

$(y_1, \ldots, y_n) \leftarrow \mathsf{simpleZeroSharing}(n)$
$(z_1, \ldots, z_n) \leftarrow (x_1 + y_1, \ldots, x_n + y_n)$
return $(z_1, \ldots, z_n)$

Algorithm 6 simpleZeroSharing

Input: $n \geq 2$
Output: n-sharing $(r_1, \ldots, r_n)$ such that $r_1 + \cdots + r_n = 0$

$r_1, \ldots, r_{n-1} \leftarrow \$$
$s \leftarrow -r_1$
for $i \leftarrow 2$ **to** $n - 1$ **do**
    $s \leftarrow s - r_i$
return $(r_1, \ldots, r_{n-1}, s)$

Algorithm 7 NlogNRefresh: log refresh gadget

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x$

$(y_1, \ldots, y_n) \leftarrow \mathsf{logZeroSharing}(n)$
$(z_1, \ldots, z_n) \leftarrow (x_1 + y_1, \ldots, x_n + y_n)$
return $(z_1, \ldots, z_n)$

Algorithm 8 logZeroSharing

Input: $n \geq 2$
Output: n-sharing $(r_1, \ldots, r_n)$ such that $r_1 + \cdots + r_n = 0$

if $n = 2$ **then**
  $r \leftarrow \$$
  return $(r, -r)$
else if $n = 3$ **then**
  $r_0, r_1 \leftarrow \$$
  return $(r_1, -(r_0 + r_1), r_0)$
else
  $m \leftarrow \lfloor n/2 \rfloor$
  $(a_1, \ldots, a_m) \leftarrow \mathsf{logZeroSharing}(m)$
  $(b_1, \ldots, b_{n-m}) \leftarrow \mathsf{logZeroSharing}(n - m)$
  $t_1, \ldots, t_m \leftarrow \$$
  $(a'_1, \ldots, a'_m) \leftarrow (a_1 + t_1, \ldots, a_m + t_m)$
  $(b'_1, \ldots, b'_m) \leftarrow (b_1 - t_1, \ldots, b_m - t_m)$
  if $m < n - m$ **then**
    $b'_{n-m} \leftarrow b_{n-m}$
  return $(a'_1, \ldots, a'_m, b'_1, \ldots, b'_{n-m})$

Algorithm 9 circular refresh gadget

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x$

$(y_1, \ldots, y_n) \leftarrow \mathsf{circZeroSharing}(n)$
$(z_1, \ldots, z_n) \leftarrow (x_1 + y_1, \ldots, x_n + y_n)$
return $(z_1, \ldots, z_n)$

Algorithm 10 circZeroSharing

Input: $n \geq 2$
Output: n-sharing $\mathbf{s} = (s_1, \ldots, s_n)$

if $n = 2$ **then**
  $r \leftarrow \$$
  return $(r, -r)$
else
  $r_1, \ldots, r_n \leftarrow \$$
  for $i \leftarrow 1$ **to** n **do**
    $s_i \leftarrow r_i - r_{(i \bmod n)+1}$
  return $(s_1, \ldots, s_n)$

Algorithm 11 Mult

Input: Shares $\boldsymbol{X} = (\boldsymbol{X}_{ij})_{i,j=1,\ldots,n}$ and $\boldsymbol{Y} = (\boldsymbol{Y}_{ij})_{i,j=1,\ldots,n}$.
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = \sum_{i=1}^{n} \sum_{j=1}^{n} \boldsymbol{X}_{ij} \cdot \boldsymbol{Y}_{ij}$

for $i \leftarrow 1$ **to** n **do**
    $z_i \leftarrow \boldsymbol{X}_{ii} \cdot \boldsymbol{Y}_{ii}$
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $r \leftarrow \$$
        $z_i \leftarrow z_i \oplus (\boldsymbol{X}_{ij} \cdot \boldsymbol{Y}_{ij} \oplus r)$
        $z_j \leftarrow z_j \oplus (\boldsymbol{X}_{ji} \cdot \boldsymbol{Y}_{ji} \oplus r)$

Algorithm 12 ISWMult

Input: n-sharings $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x \cdot y$

for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
        $x_{ij} \leftarrow x_i$
        $y_{ij} \leftarrow x_j$
$[\![z]\!] \leftarrow \mathsf{Mult}((x_{ij})_{i,j=1,\ldots,n}, (y_{ij})_{i,j=1,\ldots,n})$

Algorithm 13 MatRef (for some Refresh algorithm)

Input: Sharings $[\![x]\!] = (x_1, \ldots, x_m)$ and $[\![y]\!] = (y_1, \ldots, y_n)$
Output: Shares $\boldsymbol{X} = (\boldsymbol{X}_{ij})_{i,j=1,\ldots,m}$ and $\boldsymbol{Y} = (\boldsymbol{Y}_{ij})_{i,j=1,\ldots,n}$ such that $x \cdot y = \sum_{i=1}^{m} \sum_{j=1}^{n} \boldsymbol{X}_{ij} \cdot \boldsymbol{Y}_{ij}$

if $m = n = 1$ **then**
    return $(x_{11}), (y_{11})$
else
    $[\![\boldsymbol{x}'_1]\!], [\![\boldsymbol{x}'_2]\!] \leftarrow (x_1, \ldots, x_{\lfloor m/2 \rfloor}), (x_{\lfloor m/2 \rfloor + 1}, \ldots, x_m)$
    $[\![\boldsymbol{y}'_1]\!], [\![\boldsymbol{y}'_2]\!] \leftarrow (y_1, \ldots, y_{\lfloor n/2 \rfloor}), (y_{\lfloor n/2 \rfloor + 1}, \ldots, y_n)$
    for $i \leftarrow 1$ **to** 2 **do**
        for $j \leftarrow 1$ **to** 2 **do**
            if $\boldsymbol{x}'_i \neq () \wedge \boldsymbol{y}'_j \neq ()$ **then**
                $([\![\boldsymbol{x}''_{ij}]\!], [\![\boldsymbol{y}''_{ij}]\!]) \leftarrow \mathsf{MatRef}(\mathsf{Refresh}([\![\boldsymbol{x}'_i]\!]), \mathsf{Refresh}([\![\boldsymbol{y}'_j]\!]),)$
            else
                $[\![\boldsymbol{x}''_{ij}]\!] \leftarrow ()$
                $[\![\boldsymbol{y}''_{ij}]\!] \leftarrow ()$
return $\begin{pmatrix} [\![\boldsymbol{x}''_{11}]\!] & [\![\boldsymbol{x}''_{12}]\!] \\ [\![\boldsymbol{x}''_{21}]\!] & [\![\boldsymbol{x}''_{22}]\!] \end{pmatrix}, \begin{pmatrix} [\![\boldsymbol{y}''_{11}]\!] & [\![\boldsymbol{y}''_{12}]\!] \\ [\![\boldsymbol{y}''_{21}]\!] & [\![\boldsymbol{y}''_{22}]\!] \end{pmatrix}$

Algorithm 14 RPMult

Input: n-sharings $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x \cdot y$

$\boldsymbol{X}, \boldsymbol{Y} \leftarrow \mathsf{MatRef}([\![x]\!], [\![y]\!])$
$[\![z]\!] \leftarrow \mathsf{Mult}(\boldsymbol{X}, \boldsymbol{Y})$
return $[\![z]\!]$

Algorithm 15 Affine^f

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = f(x)$

for $i \leftarrow 1$ **to** n **do**
  if $i = 1$ **then**
    $y_i \leftarrow f(x_i)$
  else
    $y_i \leftarrow f(x_i) - f(0)$
$[\![z]\!] \leftarrow \mathsf{NlogNRefresh}([\![y]\!])$
return $[\![z]\!]$

Algorithm 16 S-box (for a multiplication gadget Mul)

Input: n-sharing $[\![x]\!] = (x_1, \ldots, x_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = \mathsf{Sbox}(x)$

$[\![a]\!] \leftarrow \mathsf{NlogNRefresh}([\![x]\!])$
$[\![b]\!] \leftarrow \mathsf{Affine}^{\cdot^2}([\![a]\!])$
$[\![c]\!] \leftarrow \mathsf{Mul}([\![a]\!], [\![b]\!])$
$[\![d]\!] \leftarrow \mathsf{Affine}^{\cdot^4}([\![c]\!])$
$[\![e]\!] \leftarrow \mathsf{NlogNRefresh}([\![c]\!])$
$[\![f]\!] \leftarrow \mathsf{Mul}([\![d]\!], [\![e]\!])$
$[\![g]\!] \leftarrow \mathsf{Affine}^{\cdot^{16}}([\![f]\!])$
$[\![h]\!] \leftarrow \mathsf{Mul}([\![d]\!], [\![g]\!])$
$[\![i]\!] \leftarrow \mathsf{NlogNRefresh}([\![h]\!])$
$[\![j]\!] \leftarrow \mathsf{Mul}([\![i]\!], [\![b]\!])$
$[\![z]\!] \leftarrow \mathsf{Affine}^{\mathsf{SbAff}}([\![j]\!])$ ▷ SbAff is the AES S-box affine tranform
return $[\![z]\!]$

Algorithm 17 Xor

Input: n-sharings $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$
Output: n-sharing $[\![z]\!] = (z_1, \ldots, z_n)$ such that $z = x \oplus y$

for $i \leftarrow 1$ **to** n **do**
  $a_i \leftarrow x_i \oplus y_i$
$[\![z]\!] \leftarrow \mathsf{NlogNRefresh}([\![a]\!])$
return $[\![z]\!]$

Algorithm 18 MixColumn

Input: n-sharings $[\![x_1]\!], \ldots, [\![x_4]\!]$
Output: n-sharings $[\![y_1]\!], \ldots, [\![y_4]\!]$ such that $(y_1, y_2, y_3, y_4) = \mathsf{MixColumn}(x_1, x_2, x_3, x_4)$

1: **for** $i \leftarrow 1$ **to** 4 **do**
2: $\quad [\![w_i]\!] \leftarrow \mathsf{NlogNRefresh}([\![x_i]\!])$
3: $\quad [\![t]\!] \leftarrow \mathsf{NlogNRefresh}([\![x_i]\!])$
4: $\quad [\![a_i]\!] \leftarrow \mathsf{Affine}^{2\times\cdot}([\![t]\!])$
5: $\quad [\![b_i]\!] \leftarrow \mathsf{Affine}^{3\times\cdot}([\![t]\!])$
6: $[\![y_1]\!] \leftarrow \mathsf{Xor}(\mathsf{Xor}([\![a_1]\!], [\![b_2]\!]), \mathsf{Xor}([\![w_3]\!], [\![w_4]\!]))$
7: $[\![y_1]\!] \leftarrow \mathsf{Xor}(\mathsf{Xor}([\![w_1]\!], [\![a_2]\!]), \mathsf{Xor}([\![b_3]\!], [\![w_4]\!]))$
8: $[\![y_1]\!] \leftarrow \mathsf{Xor}(\mathsf{Xor}([\![w_1]\!], [\![w_2]\!]), \mathsf{Xor}([\![a_3]\!], [\![b_4]\!]))$
9: $[\![y_1]\!] \leftarrow \mathsf{Xor}(\mathsf{Xor}([\![b_1]\!], [\![w_2]\!]), \mathsf{Xor}([\![w_3]\!], [\![a_4]\!]))$
10: **return** $[\![y_1]\!], [\![y_2]\!], [\![y_3]\!], [\![y_4]\!]$