# Compositional Verification of Higher-Order Masking
## Application to a Verifying Masking Compiler

Gilles Barthe[2], Sonia Belaïd[1,5], François Dupressoir[2], Pierre-Alain Fouque[4,6], and Benjamin Grégoire[3]

[1] École Normale Supérieure, Paris, France
[2] IMDEA Software Institute, Madrid, Spain
[3] Inria Sophia-Antipolis Méditerranée, France
[4] Institut Universitaire de France, France
[5] Thales Communications & Security, Gennevilliers, France
[6] Université de Rennes 1, Rennes, France

**Abstract.** The prevailing approach for building masked algorithms that can resist higher-order differential power analysis is to develop gadgets, that is, masked gates used as atomic blocks, that securely implement basic operations from the original algorithm, and then to compose these gadgets, introducing refresh operations at strategic places to guarantee that the complete circuit is protected. These compositional principles are embedded in so-called masking transformations, which are used as heuristics to achieve secure composition. Unfortunately, these transformations are seldom proved secure rigorously, and in fact, sometimes yield algorithms that are not secure against higher-order attacks. In this paper, we define a notion of *strong simulatability* that naturally supports compositional principles. Although this notion is stronger than the notion of simulatability (or perfect simulation) from previous works, we show that it is satisfied by several gadgets from the literature, including the mask refreshing gadget from Duc, Dziembowski and Faust (Eurocrypt 2014), the secure multiplication gadget from Rivain and Prouff (CHES 2010) and the secure multiplication gadget between dependent inputs from Coron et al. (FSE 2013). Then, we exploit a tight connection between strong simulatability and probabilistic information flow policies to define a (fine-grained, incremental) type system that checks (strong) simulatability of algorithms. We use the type system to validate a novel and automated transformation that outputs masked algorithms at arbitrary orders. Finally, we measure the performance of masked algorithms of AES, Keccak-f, Simon, and Speck generated by our transformation. The results are encouraging: for AES, masking at order 5, 20, and 100 respectively incur slowdowns of 100x, 750x, and x1500 w.r.t. the unmasked implementation given as input to our tool.
**Keywords:** Masking, Composition, Formal Methods, Compiler, Type System

## 1 Introduction

While most cryptosystems are assumed, or proved, to be secure against classical black-box attacks, their implementations often fail to resist *side-channel attacks* [16] which exploit, in addition to the inputs and outputs of the algorithm, the physical leakage (for example, time, power consumption or electromagnetic radiations) of the device

on which they execute. Among them, the attacks gathered under the term *Differential Power Analysis* [17] (DPA) are very powerful. These attacks exploit dependencies between the physical leakage and *sensitive values*, that depend on both secret inputs and adversarially-controlled public inputs, to recover the algorithm's secrets, sometimes from a few observation traces only.

To protect cryptographic algorithms against these attacks, Chari et al. [5] introduce the technique of *masking*, that is now widely used, in particular for protecting implementations of symmetric cryptosystems against DPA. Masking takes its inspiration from multi-party computation and more specifically from secret sharing. Secret sharing splits each sensitive intermediate variable of a program into $m$ shares in such a way that any set of $m-1$ shares is independent from the secret, which can however be recovered given all $m$ shares. Masking uses the same principles to ensure that an adversary which can make at most $t$ observations of the leakage cannot extract key-dependent information. This limitation in the number of observations is directly related to the difficulty to combine $t+1$ dependent data when they come with a sufficient level of noise. Naturally, there is a trade-off between the level of protection provided by masking, which increases with the masking order $t$, and the efficiency of masked implementations. On the one hand, the number of execution traces required to mount practical attacks and the complexity to provide concrete evaluations increase exponentially with $m$ [5,23]. On the other hand, the complexity of masked implementations increases polynomially with $t$ (since it must be that $m \geq t+1$). Nevertheless, there is a growing interest in developing methods for building higher-order masked algorithms. Ultimately, the validity of these methods is assessed by the level of protection they offer. This assessment is typically performed in two steps; first, by giving a security proof of algorithms in an appropriate model of leakage, and second, through experiments for some particular implementation on the targeted component. In this paper, we consider the first step and leave the second step for future work. With this in mind, our first task is to choose a suitable security model.

In their seminal work, Chari et al. [5] introduce the *noisy leakage model* where the adversary gets leaked values sampled according to a Gaussian distribution centered around the actual value of the sensitive variables. More recently, Prouff and Rivain [20] extend this model by considering more general noise distributions, removing the limitation on the observations' sizes and integrating the principle of Micali and Reyzin [18] according to which *only computation leaks information*. They also provide the first formal information-theoretic analysis for a whole block cipher although it relies on a leak-free component and on input and output values remaining secret. Ishai, Sahai and Wagner [15] propose another leakage model, called the $t$-threshold probing model, where the adversary can probe at most $t$ intermediate variables in a circuit without any noise. During ten years, the noisy leakage model was viewed as more relevant and close to experimental work, while the $t$-threshold probing model was extensively used to prove the security for masking schemes. However, Duc, Dziembowski and Faust [9] show that security in the noisy model of Prouff and Rivain is implied by security in the $t$-threshold model, by reducing security in the latter to the former under standard cryptographic assumptions. Recently, Duc, Faust, and Standaert [10] unveil another connection be-

tween the security of proofs in the noisy theoretical model and more concrete hardware assumptions quantified in the information theoretic framework of [22].

Thanks to these results, the $t$-threshold probing model can now realistically be used to prove the security of masked basic building blocks used in cryptographic algorithms, including for instance multipliers and S-boxes. However, proving the security of completely masked algorithms, such as AES or Keccak, remains a challenge. Existing works apply compositional principles to derive the security of a masked algorithm from the security of its components, but these works have two main shortcomings. Some of them lack a rigorous justification, and as a consequence the resulting masked algorithms are sometimes insecure; for instance, the mask refreshing operation proposed by Rivain and Prouff [21] leads to an insecure S-box [8] when composed with other secure gadgets. The others tend to overprotect the algorithms which might affect their performances; for instance, [20,9] propose to insert refresh gadgets between each operation or each time a sensitive variable is reused. Doing so, they prove the security of their algorithms but leave open the possibility of achieving the same goal with a smaller number of refresh gadgets.

**Our Contributions.** Our first contribution is definitional: starting from the observation that *simulatability*[7] used by Rivain and Prouff [21] is not sufficient to guarantee that the composition of masked algorithms remains secure, we define *strong simulatability*. Intuitively, simulatability ensures that, any set of $d \leq t$ observations made by the adversary can be simulated with at most $d$ inputs while strong simulatability ensures that the number of input shares necessary to simulate the adversary observations should be independent from the number of observations made on output wires.

We validate our definition of strong simulatability through two main theoretical contributions: first, we prove that several gadgets from the literature are strongly simulatable: the multiplication of Rivain and Prouff [21], the (same) multiplication-based mask refreshing algorithm and the multiplication between linearly dependent inputs proposed by Coron et al. in [8]. The proofs of the first and second gadgets are machine-checked in EasyCrypt [3,2], a computer-aided tool for reasoning about the security of cryptographic constructions and relational properties of probabilistic programs. The distinguishing feature of our machine-checked proofs is that they show security at arbitrary levels, whereas previous machine-checked proofs are limited to low orders (1 or 2, and in some cases up to 5). Second, we use the strongly simulatable multiplication-based mask refreshing algorithm of Rivain and Prouff to define a sound method for securely composing masked algorithms. More specifically, our main composition result shows that sensitive data can be reused as inputs to different gadgets without security flaw using judiciously placed strongly simulatable gadgets. Then, we describe a novel and efficient technique (Theorem 4) to securely compose two gadgets (without requiring that they be strongly simulatable) when the adversary can place $t$ probes inside each of them. The characteristics of our technique make it particularly well-suited to the protection of algorithms with sensitive state in the adaptive model of Ishai, Sahai and Wagner [15]. In particular, we do not require that the masking order be doubled throughout the circuit.

[7] In [21], Rivain and Prouff use the term *perfect simulation* to define this notion.

We also make two practical contributions to support the development of secure masking algorithms based on our notions and results. First, we define and implement an automated approach for verifying that an algorithm built by composing provably secure gadgets is itself secure; our approach is inspired from information-flow type systems, a thriving area in language-based security, and exploits the tight connection between (strong) simulatability and information flow policies. Second, and using advanced tools from programming languages, we implement an algorithm that takes as input an unprotected program $P$ and an arbitrary order $t$ and automatically outputs a functionally equivalent algorithm that is protected at order $t$, inserting mask refreshing gadgets where required.

Finally, our last contribution is experimental; using our transformation, we generate secure (for selected orders up to 20) masked algorithms for AES, Keccak, Simon, and Speck, and evaluate their running time.

In summary, our main contributions are: (i.) the notion of strong simulatability, and sound compositional principles for reasoning about security of masked algorithms; (ii.) proofs that existing gadgets are strongly simulatable at arbitrary order; (iii.) an automated method for verifying for any order that complete algorithms (built by composing gadgets) is secure; (iv.) a source-to-source transformation that takes an unmasked algorithm and outputs an algorithm protected at order $t$ (where $t$ is arbitrary and chosen by the user); (v.) experimental evaluation which shows that the generated algorithms behave reasonably well.

**Related work.** There has been significant work on building secure implementations of S-boxes and other core functionalities; however, most of the work is based on the weaker notion of simulatability and is justified with pen-and-paper proofs of security [15,21,14,8,6]) which remain hard to verify without formal tools. Only Faust et al. in [13] consider formally this problem in a restricted version of the noisy leakage model. In contrast, there has been relatively little work on developing automated tools for checking that an implementation is correctly masked, or for automatically producing a masked implementation from an unprotected program. This practical line of work was first considered in the context of first-order boolean masking [19]. Subsequent works extend this approach to accommodate higher-order and arithmetic masking, using type systems and SMT solvers [4], or model counting and SMT solvers [12,11]. The algorithmic complexity of the latter approach severely constrains its applications; in particular, tools based on model counting can only analyze first or second order masked implementations, and can only deal with round-reduced versions of the algorithms; for instance, only analyzing a single round of Keccak. Another recent alternative [1] exploits the tight connection between simulatability and non-interference and develops efficient algorithms for analyzing the security of masked implementations in the threshold probing model. Their approach outperforms previous work and can analyze first-order masked implementations of full AES, second-order masked implementations of round-reduced AES (4 rounds), and several third and fourth-order masked implementations of S-boxes. However, their work does not address the problem of composition and does not readily scale to higher orders or to larger algorithms. In contrast, our method scales to com-

plete algorithms at arbitrary orders, and is implemented as a source to source program transformation.

**Outline.** We first define our new security property of strong simulatability and some useful lemmas to achieve composition in Section 2. Then, we prove in Section 3 the strong simulatability of three gadgets which form the cornerstone of secure implementations: a multiplication-based refresh algorithm, the widely used secure multiplication and a multiplication between linearly dependent data. In Section 4, we informally describe our method to securely compose gadgets when the adversary can observe $t$ intermediate variables in the whole circuit. We also give a complete and compositional proof of security for the well-known inversion algorithm in the Rijndael field. In Section 5, we explain the principle of our type system to prove the security of large circuits. In Section 6, we introduce a new method to securely compose two circuits when the adversary may place $t$ probes in each of them. Finally, in Section 7, we evaluate the practicality of our approach by generating secure implementations from unprotected versions and measuring verification statistic and the performance of the resulting masked programs.

## 2 Composition

In this section, we review existing concepts from the literature and we display the current issues with composition. We then present an advantageous way of composing affine gadgets and introduce a new and useful notion which forms the basis of sound compositional reasoning: $t$-strong non-interference.

### 2.1 Gadgets

We consider programs that operate over a single finite structure $(\mathbb{K}, 0, 1, \oplus, \ominus, \odot)$; however, our techniques and tools extend smoothly to more complex scenarios. Widely used examples of such structures include binary fields, boolean rings, and modular arithmetic structures. In particular cases, it may be beneficial to consider group structures rather than rings (or fields), but we do not do so in this paper.

We define gadgets as probabilistic functions that return, in addition to their actual output, all intermediate values (including inputs) obtained during the computation. For example, the order 1 mask refreshing gadget shown in Algorithm 1 can be interpreted as the following probabilistic function where $r$ is sampled uniformly at random in $\mathbb{K}$.

$$\mathsf{Refresh}_1(a_0, a_1) = ((a_0, a_1, r), (a_0 \oplus r, a_1 \ominus r))$$

where $(a_0, a_1, r)$ is the gadget's leakage (which depends on its implementation), and $(a_0 \oplus r, a_1 \ominus r)$ is the gadget's output.

**Definition 1 (Gadgets).** *Let $m, n, \ell, o \in \mathbb{N}$. An $(m, n, \ell, o)$-gadget is a probabilistic function $G : (\mathbb{K}^m)^n \to \mathbb{K}^\ell \times (\mathbb{K}^m)^o$. Parameter $m$ denotes the gadget's* bundle size *(the number of shares over which values in $\mathbb{K}$ are shared), $n$ is the gadget's* arity *(or number of input bundles), $\ell$ is the gadget's number of* leakage wires *(or side-channel wires), and $o$ is the number of* output bundles*.*

**Algorithm 1** Example of a Gadget

1: **function** $\mathsf{Refresh}_1(\boldsymbol{a})$
2: $\quad r \xleftarrow{\$} \mathbb{K}$
3: $\quad c_0 \leftarrow a_0 \oplus r$
4: $\quad c_1 \leftarrow a_1 \ominus r$
5: $\quad$ **return** $\boldsymbol{c}$

By convention, we will always include *input wires* in a gadget's leakage (and therefore always have $m \cdot n \leq \ell$), but will never count *output wires* in a gadget's leakage (to avoid double counting). For example, function $\mathsf{Refresh}_1$ (Algorithm 1) is a $(2, 1, 3, 1)$-gadget. In the following, we fix $m$ and omit it unless relevant, and write $\widetilde{\mathbb{K}}$ for $\mathbb{K}^m$. We also assume that leakage wires are numbered following program order where relevant.

In practice, adversaries are never given direct access to the raw gadgets, but rather to *projections* of the gadget, that restrict the number of leaks and outputs the adversary can observe each time he queries the oracle. For any $\mathcal{O} \subseteq \mathbb{N}$, we define the projector $\pi_{\mathcal{O}}$ that, given a tuple, projects out the positions indicated by $\mathcal{O}$. For example, $\pi_{\{0,2\}}(x, y, z) = (x, z)$. We generalize as expected to projections on tuples of tuples.

### 2.2 $t$-simulatability and $t$-non-interference

Our goal is to prove the security of a gadget in the $t$-threshold probing model of Ishai, Sahai and Wagner [15]. To this end, we first define a notion of *$t$-non-interference* for gadgets that is sufficient to prove security in their stateless model. (We treat stateful oracles later on, in Section 5 and Appendix C.)

A *set of observations* is represented by a pair $(\mathcal{I}, \mathcal{O})$ such that $\mathcal{I} \subseteq \mathbb{N}$ is the set of *internal observations* and $\mathcal{O} \subseteq \mathbb{N}^*$ is the set of *output observations*, that is, a set of sets of indices describing which wires are observed for each output bundle; in the remainder, we often use $\Omega$ to denote observation sets. Given an $(n, \ell, o)$-gadget $G$, we say that a set of observations $(\mathcal{I}, \mathcal{O})$ is *admissible for $G$* whenever $\mathcal{I} \subseteq [0..\ell)$, $\mathcal{O} = (\mathcal{O}_0, \ldots, \mathcal{O}_{o-1})$, and $\mathcal{O}_i \subseteq [0..m)$ for every $i = 0, \ldots, o-1$. We say that $(\mathcal{I}, \mathcal{O})$ is *$t$-admissible for $G$* if it is admissible for $G$ and $|\mathcal{I}| + \sum_{i=0}^{o-1} |\mathcal{O}_i| \leq t$. Intuitively, a set of observations is admissible if only existing wires are observed, and is $t$-admissible if it is admissible and the total number of wires observed is bounded by $t$. We omit $G$ when clear from context, and abuse notation, often writing $|(\mathcal{I}, \mathcal{O})|$ to denote the quantity $|\mathcal{I}| + \sum_{i=0}^{0-1} |\mathcal{O}_i|$.

An input projection is a set $\mathcal{S} \subseteq \mathbb{N}^*$. Given an $(n, \ell, o)$-gadget $G$, we say that an input projection $\mathcal{S}$ is *compatible for $G$* whenever $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_{n-1})$ and $\mathcal{S}_i \subseteq [0..m)$ for $i = 0, \ldots, n-1$. As before, we say that an input projection $\mathcal{S}$ is *$t$-compatible for $G$* if it is compatible for $G$ and is such that $|\mathcal{S}_i| \leq t$. Intuitively, an input projection is compatible if it only projects existing input wires, and is $t$-compatible if it is compatible and projects at most $t$ wires of *each* input bundle. Here too, we abuse notation and write $|\mathcal{S}|$ to denote the quantity $\sum_{i=0}^{n-1} |\mathcal{S}_i|$

**Definition 2** **($(\mathcal{S}, \Omega)$-Simulation, -Non-Interference).** *Let $G$ be an $(n, \ell, o)$-gadget, $\mathcal{S}$ be a compatible input projection, and $\Omega$ be an admissible observation set.*

1. *We say that $G$ is* $(\mathcal{S}, \Omega)$-simulatable *(or* $(\mathcal{S}, \Omega)$-SIM*) if there exists a simulator $G_\Omega$ such that $\pi_\Omega \circ G = G_\Omega \circ \pi_\mathcal{S}$.*
2. *We say that $G$ is* $(\mathcal{S}, \Omega)$-non-interfering *(or* $(\mathcal{S}, \Omega)$-NI*) if for any $\boldsymbol{s}_0$, $\boldsymbol{s}_1$ in $\widetilde{\mathbb{K}}^n$ such that $\pi_\mathcal{S}(\boldsymbol{s}_0) = \pi_\mathcal{S}(\boldsymbol{s}_1)$, we have $\pi_\Omega \circ G(\boldsymbol{s}_0) = \pi_\Omega \circ G(\boldsymbol{s}_1)$.*

**Lemma 1 (Simulation $\Leftrightarrow$ Non-Interference).** *For all $(n, \ell, o)$-gadget $G$, compatible $\mathcal{S}$ and admissible $\Omega$, $G$ is $(\mathcal{S}, \Omega)$-SIM iff $G$ is $(\mathcal{S}, \Omega)$-NI.*

We now define $t$-NI , which is equivalent to the $(\mathcal{I}, \mathcal{O})$-non-interference notion of Barthe et al. [1], and therefore characterizes security in the $t$-threshold probing model.

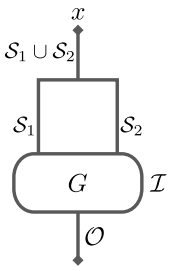
**Definition 3 ($t$-Non-Interference).** *An $(n, \ell, o)$-gadget $G$ is* $t$-non-interfering *iff, for every $t$-admissible $\Omega$, there exists a $t$-compatible $\mathcal{S}$ such that $G$ is $(\mathcal{S}, \Omega)$-NI. (Equivalently, $G$ is $(\mathcal{S}, \Omega)$-SIM.)*

Inspecting proofs of security from the literature reveals that gadgets often satisfy a slightly stronger property, which we find convenient to introduce as it is more clearly suited to compositional reasoning, and is critically used in the definition of robust composition (Section 6).

**Definition 4 ($t$-Tight Non-Interference).** *We say that an $(n, \ell, o)$-gadget $G$ is* $t$-tightly non-interfering *($t$-TNI) whenever for any $t$-admissible observation set $\Omega$ on $G$, there exists a $|\Omega|$-compatible $\mathcal{S}$ such that $G$ is $(\mathcal{S}, \Omega)$-NI.*

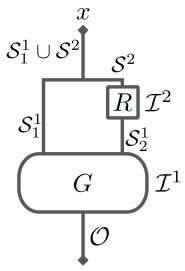
### 2.3 Issues with composition

We justify the need for intuitive and correct composition results with simple examples, that also serve to illustrate the need for stronger simulation properties and the basic principles of our verification techniques. We consider the circuits shown in Figures 1 and 2, that compute $G(x, x)$ for some shared variable $x$ and some $(2, \ell, 1)$-gadget $G$. $R$ is some $(1, \ell', 1)$-gadget implementing the identity function.




with $|\mathcal{I}| + |\mathcal{O}| \leq t$.

Fig. 1: Diagram 1




with $|\mathcal{I}^2| + |\mathcal{I}^1| + |\mathcal{O}| \leq t$.

Fig. 2: Diagram 2

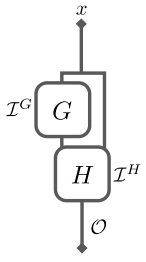



Fig. 3: Affine-NI composition.

Let us first consider the circuit shown in Diagram 1 (Figure 1), assuming that $G$ is $t$-NI (for example, $G$ could be Rivain and Prouff's secure multiplication gadget [21]), and that the adversary makes a $t$-admissible set of observations $\Omega = (\mathcal{I}, \mathcal{O})$.

From the fact that $G$ is $t$-NI, we know that $\Omega$ can be perfectly simulated using shares $\mathcal{S}_1$ of its first input and shares $\mathcal{S}_2$ of its second input. This means that the simulator for the entire circuit must be given shares $\mathcal{S}_1 \cup \mathcal{S}_2$ of $x$. The problem is of course that the number of shares of $x$ needed to simulate the full circuit cannot be proved, in general, to be less than $t+1$, since we only know that $|\mathcal{S}_1 \cup \mathcal{S}_2| < 2t+1$. A similar argument applies when $G$ is $t$-TNI.

However, the circuit from Figure 1 is secure under some specific conditions on $G$. One such condition, which we call the affine condition, states that the gadget $G$ can be simulated in such a way that $\mathcal{S}_1 = \mathcal{S}_2$. In this case, the circuit is therefore $t$-NI.

Assume now that $G$ can be simulated using a number of shares of its inputs that is bounded by the number of observations made *in* $G$ by the adversary, rather than by the number of observations made in the entire circuit (that is, $t$). In other words, we now know that $|\mathcal{S}_1|, |\mathcal{S}_2| \leq |\mathcal{I}| + |\mathcal{O}|$. This is obviously still insufficient to guarantee that the whole circuit can be simulated without knowledge of $x$, since the simulator could need up to $2(|\mathcal{I}| + |\mathcal{O}|)$ shares of $x$. However, if we consider the circuit shown in Diagram 2 (Figure 2), we can prove that the circuit is secure under some conditions on $R$. One such condition, which we call the strong simulatability condition, states that the input of gadget $R$ can be simulated in such a way that $|\mathcal{S}^2| \leq |\mathcal{I}^2|$ whenever $|\mathcal{I}^2 \cup \mathcal{S}_2^1| \leq t$. Under this assumption, we can prove that the whole circuit is $t$-NI. Indeed, in such a setting, the entire circuit can be simulated using at most $|\mathcal{S}_1^1 \cup \mathcal{S}^2|$ shares of $x$. Since $|\mathcal{S}_1^1| \leq |\mathcal{I}^1| + |\mathcal{O}|$, $|\mathcal{S}^2| \leq |\mathcal{I}^2|$ and $|\mathcal{I}^2| + |\mathcal{I}^1| + |\mathcal{O}| \leq t$, then $|\mathcal{S}_1^1 \cup \mathcal{S}^2| \leq t$.

The next two paragraphs formalize these conditions. For clarity, we only consider gadgets that output a single wire bundle and omit the value of $o$ in descriptions. We generalize our reasoning to arbitrary gadgets in Section 5.

### 2.4 Affine Non-Interference

We define the class of affine gadgets and show that they satisfy useful security properties which can be used to build efficient compositions. Informally, a gagdet is affine if it performs sharewise computations. By extension, the class of affine functions is the class of functions that can be computed using affine gadgets. For example, using boolean masking in $\mathbb{K} = \mathsf{GF}(2^n)$, affine functions include linear operators (field addition, shifts and rotations), bitwise negation and scalar multiplication.

**Definition 5 (Affine Gadgets).** *A $(n, \ell)$-gadget $G$ is said to be* affine *whenever there exists a family $(G_i)_{0 \leq i < m}$ of probabilistic functions $G_i \in \mathbb{K}^n \to \mathbb{K}^{\ell_i} \times \mathbb{K}$ such that $\sum_{i=0}^{m-1} \ell_i = \ell$, and $G = \prod_{i=0}^{m-1} G_i$ (where the product on functions reorders outputs so that side-channels appear first and in the right order).*

Intuitively, an affine gadget $G$ can be seen as the product (or parallel composition) of $m$ gadgets $(G_i)_{0 \leq i < m}$ such that $G_i$ takes as input the $i^{th}$ share of each of $G$'s $n$ inputs and outputs the $i^{th}$ share of $G$'s output. It is important to note that the leakage computation should also be distributed. Note that the composition of affine gadgets is affine.

Affine gadgets fulfill a more precise property that partially specifies the set of shares of each input required to simulate a given set of observations. We define this more precise property as *affine non-interference*.

**Definition 6 (Affine-Non-Interference).** *An $(n, \ell)$-gadget $G$ is affine-NI iff for every admissible set of observations $\Omega = (\mathcal{I}, \mathcal{O})$, there exists a $|\mathcal{I}|$-compatible $\widehat{\mathcal{S}}$ such that $G$ is $(\widehat{\mathcal{S}} \cup \mathcal{O}, \Omega)$-NI.*

We now show that every affine gadget is affine-NI, and illustrate how this precise property can be used to prove $t$-NI in contexts where other gadget-level properties may be insufficient.

**Theorem 1.** *Every affine $(n, \ell)$-gadget $G$ is affine-NI.*

*Proof.* We prove that $G$ is $(\widehat{\mathcal{S}} \cup \mathcal{O}, \Omega)$-SIM instead. The simulator $G_\Omega$ can simply be constructed as the product of all the $G_i$ components of $G$ in which adversary observations occur. This requires the $\mathcal{O}$ shares, plus at most $|\mathcal{I}|$ additional ones due to observations on side-channels. Intuitively, internal observations are simulated using shares indexed by $\widehat{\mathcal{S}}$, and output observations are simulated using shares indexed by $\mathcal{O}$. □

We now consider a simple example which allows us to see how the details of affine-NI allow fine-grained compositional security proofs.

**Lemma 2 (Composition of affine-NI gadgets).** *Assuming both $G$ and $H$ are affine-NI, the circuit shown in Figure 3, is affine-NI.*

*Proof.* Let $\Omega = ((\mathcal{I}^G, \mathcal{I}^H), \mathcal{O})$ be a $t$-admissible observation set for the circuit shown in Figure 3. By affine-NI of $H$, we know that there exists $\widehat{\mathcal{S}}^H$ such that $|\widehat{\mathcal{S}}^H| \leq |\mathcal{I}^H|$ and $H$ is $((\widehat{\mathcal{S}}^H \cup \mathcal{O}, \widehat{\mathcal{S}}^H \cup \mathcal{O}), (\mathcal{I}^H, \mathcal{O}))$-NI. It is therefore sufficient for us to simulate shares $\widehat{\mathcal{S}}^H \cup \mathcal{O}$ of each of $H$'s inputs. One of them is given by $G$'s outputs. By affine-NI of $G$, we know that there exists $\widehat{\mathcal{S}}^G$ such that $|\widehat{\mathcal{S}}^G| \leq |\mathcal{I}^G|$ and $G$ is $(\widehat{\mathcal{S}}^G \cup (\widehat{\mathcal{S}}^G \cup \mathcal{O}), (\mathcal{I}^G, \widehat{\mathcal{S}}^G \cup \mathcal{O}))$-NI. It is therefore sufficient for us to simulate shares $\widehat{\mathcal{S}}^G \cup \widehat{\mathcal{S}}^H \cup \mathcal{O}$ of $G$'s input. Note that $G$'s input is also $H$'s second input. We therefore need shares $(\widehat{\mathcal{S}}^G \cup (\widehat{\mathcal{S}}^H \cup \mathcal{O})) \cup (\widehat{\mathcal{S}}^H \cup \mathcal{O})$ to simulate the whole circuit. Simplifying, we obtain that we need $\widehat{\mathcal{S}}^G \cup \widehat{\mathcal{S}}^H \cup \mathcal{O}$ shares of the circuit's inputs to simulate the whole circuit. By the constraint on the sizes of $\widehat{\mathcal{S}}^G$ and $\widehat{\mathcal{S}}^H$ and the $t$-admissibility constraint on $\Omega$, we deduce that the circuit can be simulated using at most $t$ shares of its input and conclude. □

Note that the precise expression for the set of input shares used to simulate leakage given by the notion of affine-NI is key in finishing this proof. In particular, this fine-grained example of composition would be impossible without knowing that the sets of shares of each of $H$'s inputs required to simulate $H$ are equal, and without knowing, in addition, that the set of shares of $G$'s inputs required to simulate $G$ is an extension of the set of shares of its outputs we need to simulate.

### 2.5 $t$-Strong Non-Interference

We now introduce our main notion for which we will derive sound and secure composition principles: *$t$-strong non-interference*.

**Definition 7 ($t$-Strong Non-Interference).**
*A $(n, \ell)$-gadget $G$ is said to be* $t$-strongly non-interfering *(or $t$-SNI) whenever, for every $t$-admissible $\Omega = (\mathcal{I}, \mathcal{O})$, there exists a $|\mathcal{I}|$-compatible $\mathcal{S}$ such that $G$ is $(\mathcal{S}, \Omega)$-NI.*

Essentially, a $t$-SNI gadget can be simulated using a number of each of its input shares that is only bounded by the number of observations made by the adversary on inner leakage wires, and is independent from the number of observations made on output wires, as long as the total number of observations does not exceed $t$. This independence with the output observations is critical in securely composing gadgets with related inputs. More specifically, the following lemma illustrates how $t$-SNI supports compositional reasoning.

**Lemma 3 (Composition of $t$-SNI).** *If $G$ and $R$ are $t$-SNI, the circuit shown in Figure 2 is $t$-SNI.*

*Proof.* Let $\Omega = (\mathcal{I}, \mathcal{O})$ be a $t$-admissible set of observations on the circuit, where $\mathcal{I} = (\mathcal{I}^1, \mathcal{I}^2)$ is partitioned depending on the subgadget in which observations occur. The set $\Omega^1 = (\mathcal{I}^1, \mathcal{O})$ of observations made on $G$ is $t$-admissible, and we therefore know, by $t$-SNI, that there exists an —$\mathcal{I}^1$—-compatible input projection $\mathcal{S}^1 = (\mathcal{S}_1^1, \mathcal{S}_2^1)$ for $G$ such that $G$ is $(\mathcal{S}, \Omega^1)$-NI. Simulating $G$ and considering that simulator part of the adversary, the set of adversary observations made on $R$ is now $\Omega^2 = (\mathcal{I}^2, \mathcal{S}_2^1)$. Since $|\mathcal{S}_2^1| \leq |\mathcal{I}^1|$, we know that $\Omega^2$ is $t$-admissible and can make use of the fact that $R$ is $t$-SNI. This yields the existence of a $|\mathcal{I}^2|$-compatible input projection $\mathcal{S}^2$ for $R$ such that $R$ is $(\mathcal{S}^2, \Omega^2)$-NI. We now have a simulator for the whole circuit that makes use of the set of shares $\mathcal{S}_1^1 \cup \mathcal{S}^2$ of $x$. Since $|\mathcal{S}_1^1| \leq |\mathcal{I}^1|$, $|\mathcal{S}^2| \leq |\mathcal{I}^2|$, and $|\mathcal{I}^1| + |\mathcal{I}^2| \leq t$, we can conclude. Note that this proof method also constructs the simulator for the circuit by composing the simulators for the core gadgets after checking that they exist. □

An interesting example, based on Rivain and Prouff's algorithm to compute an inverse in $\mathsf{GF}(2^8)$ is discussed in Section 3 (Figure 4 and Theorem 2).

*A remark on efficiency.* An unexpected benefit of strong non-interference is that it leads to significant efficiency gains: specifically, one can safely dispense from refreshing the output of a strongly non-interfering gadget. We exploit this insight to improve the efficiency of algorithms transformed by our compiler.

## 3 Some Useful SNI Gadgets

In this section, we show that the mask refreshing gadget from Rivain and Prouff [21] does not satisfy $t$-SNI, and hence should be used very carefully (or not at all), whereas the refreshing gadget from Duc, Dziembowski and Faust [9] satisfies $t$-SNI and thus can be used to compositionally define masked algorithms. Then, we show that the multiplication gadget provided by Rivain and Prouff in [21] is also $t$-SNI. Finally, we also show that the gadget proposed by Coron et al. [8] to compute $h : x \mapsto x \otimes g(x)$ for some linear function $g$ and some internal, associative $\otimes$ that distributes over addition in $\mathbb{K}$ is also $t$-SNI.

The proofs for the mask refreshing and multiplication gadgets have been verified formally in EasyCrypt; for convenience, we also provide pen-and-paper proof sketches for these proofs and that of the combined gadget from [8] in Appendix B.

### 3.1 Mask Refreshing Algorithms

We consider here two mask refreshing algorithms: the RefreshMasks algorithm introduced in [21], that consists in adding a uniform sharing of 0, and an algorithm based on multiplying by a trivial sharing of 1 using secure multiplication algorithm SecMult [21]. Below, we show that the former is not $t$-SNI, whilst the latter is $t$-SNI and can therefore be used to compositionally build secure implementations.

**Addition-Based Mask Refreshing Algorithm.** Algorithm 2 presents the addition-based refreshing algorithm introduced by Rivain and Prouff [21]. Since it only samples

**Algorithm 2** Addition-Based Mask Refreshing Algorithm

```
1: function RefreshMasks(a)
2:     for i = 1 to t do
3:         u ←$ 𝕂
4:         a_0 ← a_0 ⊕ u
5:         a_i ← a_i ⊕ u
6:     return a
```

a number of random masks linear in the masking order, this algorithm is very efficient, and is in fact $t$-NI (as proved by its authors [21]). However, it is not $t$-SNI. Indeed, for any order $t \geq 2$, observing the intermediate variable $a_0 \oplus u$ and the output $a_1 \oplus u$ (in the first loop iteration) lets the adversary learn the sum $a_0 \oplus a_1$, which cannot be perfectly simulated from less than two of $a$'s shares. Since it is not $t$-SNI, the RefreshMasks algorithm cannot be used as illustrated in Section 2 to compositionally guarantee the security of masked algorithms. This explains, in particular, the flaw it induces [8] when used in Rivain and Prouff's secure S-box algorithm [21]. In fact, the counterexample we gave to $t$-SNI is central to the flaw exhibited by Coron et al. [8].

**Multiplication-Based Mask Refreshing Algorithm.** Algorithm 3 presents the mask refreshing algorithm by Duc et al. [9], based on applying the secure multiplication of Rivain and Prouff [21] to a trivial sharing of 1 as $(1, 0, \ldots, 0)$.

**Proposition 1.** *Algorithm 3 is $t$-SNI.*

The proof is given in Appendix B. In the following, we call $\mathsf{Refresh}_t$ (omitting the index when clear from context) the core gadget implemented using Algorithm 3.

**Algorithm 3** Multiplication-Based Mask Refreshing Algorithm

**function** RefreshMult($\boldsymbol{a}$)
  **for** $i = 0$ **to** $t$ **do**
    $c_i \leftarrow a_i$
  **for** $i = 0$ **to** $t$ **do**
    **for** $j = i + 1$ **to** $t$ **do**
      $r \xleftarrow{\$} \mathbb{K}$ /* this random value is referred to as $r_{i,j}$ */
      $c_i \leftarrow c_i \oplus r$ /* the result is referred to as $c_{i,j}$ */
      $c_j \leftarrow c_j \ominus r$ /* the result is referred to as $c_{j,i}$ */
  **return** $\boldsymbol{c}$

### 3.2 Secure Multiplication Algorithms

We focus here on two multiplication algorithms: the SecMult algorithm introduced in [21] and Algorithm 4 from [8], which computes function $h : x \mapsto x \odot g(x)$ for some linear function $g$.

**SecMult algorithm.** We show Rivain and Prouff's multiplication algorithm SecMult in Algorithm 4 [21]. Note that this algorithm is correct and secure for the computation of any internal, associative and commutative operation $\otimes$ that distributes over addition in $\mathbb{K}$ ($\oplus$). This includes, for example, field multiplication $\odot$ in $\mathsf{GF}(2^n)$, and multiplication $\&$ in the boolean ring $\mathbb{B}^n$. In addition to being $t$-NI as claimed by Rivain and Prouff [21],

**Algorithm 4** Secure Multiplication Algorithm [21]

**function** SecMult($\boldsymbol{a}, \boldsymbol{b}$)
  **for** $i = 0$ **to** $t$ **do**
    $c_i \leftarrow a_i \otimes b_i$
  **for** $i = 0$ **to** $t$ **do**
    **for** $j = i + 1$ to $t$ **do**
      $r \xleftarrow{\$} \mathbb{K}$ /* this random value is referred to as $r_{i,j}$ */
      $c_i \leftarrow c_i \oplus r$ /* referred to as $c_{i,j}$ */
      $r \leftarrow a_i \otimes b_j \ominus r \oplus a_j \otimes b_i$ /* referred to as $r_{j,i}$ */
      $c_j \leftarrow c_j \oplus r$ /* referred to as $c_{j,i}$ */
  **return** $\boldsymbol{c}$

we show that it is also $t$-SNI (Proposition 2) in Appendix B. This stronger security property makes it valuable for performance whilst retaining provable security, since it may reduce the number of mask refreshing gadgets required to compositionally prove security.

**Proposition 2.** *Algorithm 4 is $t$-SNI.*

**MultLinear algorithm to compute $x \otimes g(x)$.** Coron et al. [8] introduce an extended multiplication algorithm, which we recall as Algorithm 5, and prove that it is $t$-NI. In fact, their proof even shows that the gadget is $t$-TNI, but the authors do not identify this stronger property. We show here that this algorithm is in fact $t$-SNI and therefore dispenses the user from having to refresh its output's masks.

**Algorithm 5** $h : x \mapsto x \otimes g(x)$ [8, Algorithm 4]

**function** MultLinear($\boldsymbol{a}$)
  **for** $i = 0$ **to** $t$ **do**
    **for** $j = i + 1$ **to** $t$ **do**
      $r_{i,j} \xleftarrow{\$} \mathbb{K}$
      $r'_{i,j} \xleftarrow{\$} \mathbb{K}$
      $t \leftarrow a_i \otimes g(r'_{i,j}) \ominus r_{i,j}$
      $t \leftarrow t \oplus (r'_{i,j} \otimes g(a_i))$
      $t \leftarrow t \oplus (a_i \otimes g(a_j \ominus r'_{i,j})$
      $t \leftarrow t \oplus ((a_j \ominus r'_{i,j}) \otimes g(a_i))$
      $r_{j,i} \leftarrow t$ /* $r_{j,i} = r_{i,j} \oplus a_i \otimes g(a_j) \oplus a_j \otimes g(a_i)$ */
  **for** $i = 0$ **to** $t$ **do**
    $c_i \leftarrow a_i \otimes g(a_i)$
    **for** $j = 0$ **to** $t, j \neq i$ **do**
      $c_i \leftarrow c_i \oplus r_{i,j}$ /* referred to as $c_{i,j}$ */
  **return** $\boldsymbol{c}$

**Proposition 3.** *Algorithm 5 is $t$-SNI.*

A pen-and-paper proof of Proposition 3 is given in Appendix B.

## 4 Simple Compositional Proofs of Security

In this section, we show how the various notions of non-interference, and more specifically the notion of $t$-SNI can be used to obtain compositional proofs of security for large circuits. We start by abstractly describing a generic proof method for compositionally proving a circuit $t$-NI based only on affine, $t$-SNI, and $t$-TNI properties of core gadgets and checking simple arithmetic side-conditions. We then illustrate it by detailing a compositional proof of security for a masked version of an inversion algorithm in $\mathsf{GF}(2^8)$ [21].

### 4.1 Securely Composing Secure Gadgets

We consider a masked circuit $P$ constructed by composition of $n$ chosen core gadgets (affine gadgets or those $t$-SNI gadgets discussed in Section 3), and a topological ordering on $P$'s gadgets (using 1, rather than $n$, to denote the last gadget according to

that ordering). Let $\Omega$ be an arbitrary $t$-admissible observation set on $P$. We split $\Omega$ according to whether observations occur on $P$'s output bundles (we name those $\mathcal{O}^i$, one for each of $P$'s $o$ outputs) or are internal to a gadget in $P$'s (we name those $\mathcal{I}^i$, one for each core gadget in $P$). The $t$-admissibility constraint on $\Omega$ implies the following *global constraint* on its components:

$$|\Omega| = \sum_{1 \leq i \leq o} |\mathcal{O}^i| + \sum_{1 \leq i \leq n} |\mathcal{I}^i| \leq t$$

The process starts with:

- an initial set of constraint $C$ that only contains the global constraint;
- for each wire that serves as a connection between core gadgets in $P$, a set $\mathcal{O}^i_j$, that intuitively corresponds to gadget $i$'s $j$th output bundle; all $\mathcal{O}^i_j$ are initially empty except for those that correspond to $P$'s output bundles, to which the corresponding $\mathcal{O}^k$ (or union of $\mathcal{O}^k$s, if the same bundle is used multiple times as output) is assigned.

Starting from Gadget 1 (the last gadget according to the chosen topological ordering) and the initial state described above, and for each gadget $G^i$ (progressing back through the chosen ordering), the following operations are performed:

1. check that the side condition for gadget $G^i$'s non-interference property follows from the constraints in $C$:
   - if $G^i$ is $t$-TNI or $t$-SNI, check that $C \Rightarrow |\mathcal{I}^i| + \sum_{1 \leq j \leq o_i} |\mathcal{O}^i_j| \leq t$;
   - if $G^i$ is affine, the side-condition is trivial.
2. using the corresponding non-interference property, derive a set of shares for each of $G^i$'s inputs that suffice to simulate the internal and output leakage in $G^i$, and add the corresponding constraints to $C$:
   - if $G^i$ is $t$-TNI, a fresh set of indices $\mathcal{S}^i_j$ is introduced for each input bundle $j$, and the constraint $|\mathcal{S}^i_j| \leq |\mathcal{I}^i| + \sum_{1 \leq k \leq o_i} |\mathcal{O}^i_k|$ is added to $C$ for all $j$;
   - if $G^i$ is $t$-SNI, a fresh set of indices $\mathcal{S}^i_j$ is introduced for each input bundle $j$, and the constraint $|\mathcal{S}^i_j| \leq |\mathcal{I}^i|$ is added to $C$ for all $j$;
   - if $G^i$ is affine, a fresh set of indices $\widehat{\mathcal{S}}^i$ is introduced, and the constraint $|\widehat{\mathcal{S}}^i| \leq |\mathcal{I}^i|$ is added to $C$.
3. the newly computed sets of shares on $G^i$'s inputs are propagated to become output observations on the gadget from which they come (except if they correspond to $C$'s inputs):
   - if $G^i$ is $t$-TNI or $t$-SNI, for all $j$, if $G^i$'s $j$th input bundle is connected to another gadget $G^{i'}$'s $k$th output bundle, set $\mathcal{O}^{i'}_k \leftarrow \mathcal{O}^{i'}_k \cup \mathcal{S}^i_j$;
   - if $G^i$ is affine, if any of $G^i$'s input bundles is connected to another gadget $G^{i'}$'s $k$th output bundle, set $\mathcal{O}^{i'}_k \leftarrow \mathcal{O}^{i'}_k \cup \widehat{\mathcal{S}}^i \cup \mathcal{O}^i$.

If, at any point in this process, a side-condition fails to check, the circuit is not compositionally secure (although it may still be $t$-NI). However, the only case in which such a failure could occur is if a gadget's output serves as input to multiple gadgets. In this case, the failure in checking the side-condition can in fact be used to automatically insert a mask refreshing gadget as needed, and resume the proof from that point on.

Once all gadgets in the circuit are simulated as described, one can then easily check whether the accumulated constraints in $C$ are sufficient to guarantee that the set of

shares associated with each one of $P$'s input bundles is of a size smaller than or equal to $t$.

### 4.2 An Example: AES inversion algorithm by Rivain and Prouff

We now illustrate this process on Rivain and Prouff's algorithm for computing inversion in $\mathsf{GF}(2^8)$ [21,8] when implemented over $t+1$ shares. A circuit implementing this operation securely is shown in Figure 4. We use a simple composition argument to prove that this inversion is $t$-SNI, relying on the fact that the multiplication gadget $\otimes$ and the refreshing gagdet $R$ (that is, Refresh) are both $t$-SNI. We recall that the function $x \mapsto x^{2^n}$ for any $n$ is linear in binary fields and rely on affine gadgets $\cdot^2$, $\cdot^4$ and $\cdot^{16}$ to compute the corresponding (linear) functionalities.

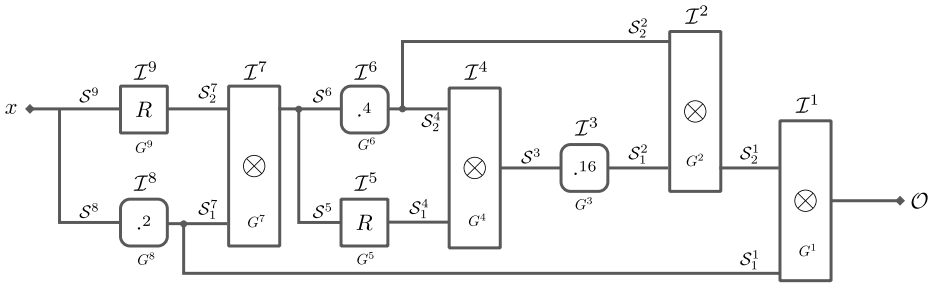



Fig. 4: Gadget $\cdot^{254}$

**Theorem 2.** *Gadget $\cdot^{254}$, shown in Figure 4, is $t$-SNI.*

*Proof.* The proof follows the process described above. We detail it here to illustrate the compositional proof process on a practical example.

Let $\Omega = (\bigcup_{1\leq i\leq 9} \mathcal{I}^i, \mathcal{O})$ be a $t$-admissible observation set. In particular, we know that the *global constraint* $|\mathcal{O}| + \sum_{1\leq i\leq 9} |\mathcal{I}^i| \leq t$ holds. The proof constructs the simulator by simulating each gadget in turn, starting from the final multiplication (Gadget 1) and progressing from right to left and upward.

**Gadget 1 -** since $\otimes$ is $t$-SNI and $|\mathcal{I}^1 \cup \mathcal{O}| \leq t$ (by the global contraint), we know that there exist observation sets $\mathcal{S}_1^1, \mathcal{S}_2^1$ such that $|\mathcal{S}_1^1| \leq |\mathcal{I}^1|$, $|\mathcal{S}_2^1| \leq |\mathcal{I}^1|$ and Gadget 1 is $((\mathcal{S}_1^1, \mathcal{S}_2^1), (\mathcal{I}^1, \mathcal{O}))$-NI; (note that $\mathcal{S}_1^1$ and $\mathcal{S}_2^1$ become output observations on Gadgets 8 and 2, respectively)

**Gadget 2 -** since $\otimes$ is $t$-SNI and $|\mathcal{I}^2 \cup \mathcal{S}_2^1| \leq t$ (by the simulation of Gadget 1 and the global constraint, we know that there exist observations sets $\mathcal{S}_1^2$, $\mathcal{S}_2^2$ such that $|\mathcal{S}_1^2| \leq |\mathcal{I}^2|$, $|\mathcal{S}_2^2| \leq |\mathcal{I}^2|$, and Gadget 2 is $((\mathcal{S}_1^2, \mathcal{S}_2^2), (\mathcal{I}^2, \mathcal{S}_2^1))$-NI;

**Gadget 3 -** since $\cdot^{16}$ is affine, we know that there exists an observation set $\mathcal{S}^3$ such that $|\mathcal{S}^3| \leq |\mathcal{I}^3| + |\mathcal{S}_1^2| \leq |\mathcal{I}^3| + |\mathcal{I}^2|$ (by the simulation of Gadget 2) and Gadget 3 is $(\mathcal{S}^3, (\mathcal{I}^3, \mathcal{S}_1^2))$-NI;

**Gadget 4 -** since $\otimes$ is $t$-SNI and $|\mathcal{I}^4 \cup \mathcal{S}^3| \leq t$ (by the simulation of Gadget 3 and the global constraint), we know that there exist observation sets $\mathcal{S}_1^4$, $\mathcal{S}_2^4$ such that $|\mathcal{S}_1^4| \leq |\mathcal{I}^4|$, $|\mathcal{S}_2^4| \leq |\mathcal{I}^4|$, and Gadget 4 is $((\mathcal{S}_1^4, \mathcal{S}_2^4), (\mathcal{I}^4, \mathcal{S}^3))$-NI;

**Gadget 5 -** since RefreshMult is $t$-SNI and $|\mathcal{I}^5 \cup \mathcal{S}_1^4| \leq t$ (by the simulation of Gadget 4 and the global constraint), we know that there exist observation sets $\mathcal{S}_1^5$, $\mathcal{S}_2^5$ such that $|\mathcal{S}_1^5| \leq |\mathcal{I}^5|$, $|\mathcal{S}_2^5| \leq |\mathcal{I}^5|$, and Gadget 5 is $((\mathcal{S}_1^5, \mathcal{S}_2^5), (\mathcal{I}^5, \mathcal{S}_1^4))$-NI;

**Gadget 6 -** since $\cdot^4$ is affine, we know that there exists an observation set $\mathcal{S}^6$ such that $|\mathcal{S}^6| \leq |\mathcal{I}^6| + |\mathcal{S}_2^2 \cup \mathcal{S}_2^4| \leq |\mathcal{I}^6| + |\mathcal{I}^2| + |\mathcal{I}^4|$ (by the simulation of Gadgets 2 and 4) and Gadget 6 is $(\mathcal{S}^6, (\mathcal{I}^6, \mathcal{S}_2^2 \cup \mathcal{S}_2^4))$-NI;

**Gadget 7 -** since $\otimes$ is $t$-SNI and $|\mathcal{I}^7 \cup \mathcal{S}^5 \cup \mathcal{S}^6| \leq t$ (by the simulation of Gadgets 5 and 6 and the global constraint), we know that there exist observation sets $\mathcal{S}_1^7$, $\mathcal{S}_2^7$ such that $|\mathcal{S}_1^7| \leq |\mathcal{I}^7|$, $|\mathcal{S}_2^7| \leq |\mathcal{I}^7|$, and Gadget 7 is $((\mathcal{S}_1^7, \mathcal{S}_2^7), (\mathcal{I}^7, \mathcal{S}^5 \cup \mathcal{S}^6))$-NI;

**Gadget 8 -** since $\cdot^4$ is affine, we know that there exists an observation set $\mathcal{S}^8$ such that $|\mathcal{S}^8| \leq |\mathcal{I}^8| + |\mathcal{S}_1^1 \cup \mathcal{S}_1^7| \leq |\mathcal{I}^8| + |\mathcal{I}^1| + |\mathcal{I}^7|$ (by the simulation of Gadgets 1 and 7) and Gadget 8 is $(\mathcal{S}^8, (\mathcal{I}^8, \mathcal{S}_1^1 \cup \mathcal{S}_1^7))$-NI;

**Gadget 9 -** since $\otimes$ is $t$-SNI and $|\mathcal{I}^9 \cup \mathcal{S}_2^7| \leq t$ (by the simulation of Gadget 7 and the global constraint), we know that there exists an observation set $\mathcal{S}^9$ such that $|\mathcal{S}^9| \leq |\mathcal{I}^9|$, and Gadget 9 is $(\mathcal{S}^9, (\mathcal{I}^9, \mathcal{S}^7))$-NI;

Each of these steps gives us the existence of a simulator for the relevant gadget. Composing them together constructs a simulator for the whole circuit that expects $|\mathcal{S}^8 \cup \mathcal{S}^9|$ shares of $x$. Since we have $|\mathcal{S}^8| \leq |\mathcal{I}^7| + |\mathcal{I}^1|$ and $|\mathcal{S}^9| \leq |\mathcal{I}^9|$, we can conclude that $|\mathcal{S}^8 \cup \mathcal{S}^9| \leq \sum_{1 \leq i \leq 9} |\mathcal{I}^i|$ and therefore that gadget $\cdot^{254}$ is $t$-SNI. □

*Remark 1.* In the proof of Theorem 2, we do not precisely keep track of the simulation sets for affine gadgets as the more general bounds on their size are sufficient to conclude. In practice, when considering large affine sub-circuits, it is often important to keep track of the precise composition of the simulation sets rather than just their size in order to avoid false negatives.

## 5 Formalization

Although proofs of $t$-NI (or $t$-SNI) using the compositional methods described in Section 2 are easier to write and check than existing proofs, they quickly get cumbersome on large programs, and simply typing sub-circuits as gadgets and reusing the results may be imprecise in general (in particular when the sub-circuit has multiple outputs). We automate the construction and verification of these compositional proofs using a type system that tracks, at every program point, the number of shares of each sensitive variable required to simulate the rest of the circuit. The judgments of the type system can express soundly and precisely the notions of affine-NI, $t$-TNI and $t$-SNI introduced in Section 2. This allows us to typecheck that a gadget is, say, affine-NI in isolation, and to use it as is, without having to retype it in context. In addition, our type system generalizes on the ideas presented in Sections 2 and 4 in several ways: i. it clearly differentiates between *sensitive* and *public* wires, which prevents many false positives and allows the selection of more efficient gadgets for some operations (for example, multiplication by a public variable in $\mathsf{GF}(2^n)$ is simply scalar multiplication); and ii. it supports gadgets with multiple outputs, which allows us in particular to deal with state.

After laying down some formal definitions, we define a notion of typing judgment that is sufficient to prove $t$-NI. We then illustrate why a type system that simply implements and automates the proof technique discussed in Section 4 does not immediately allow incrementality. This, in conjunction with the fact that simply retyping every sub-procedure in an algorithm every time it is used would not scale well, justifies the complexity of our incremental type system.

### 5.1 Language and Types

For clarity in the formalization, we consider a small language (Figure 5) that allows us to provide compact descriptions of circuits as compositions of gadgets. Note that this is done without loss of generality: variables in the language can be seen as descriptions of the wiring between gadgets that fix one particular choice of topological ordering on the circuit's gadgets. In practice, we extend the language slightly to allow for simple

$$
\begin{array}{rll}
p ::= & | \ \boldsymbol{x} \leftarrow G(\boldsymbol{y}) & \text{gadget call } (\boldsymbol{x} \text{ and } \boldsymbol{y} \text{ may be tuples}) \\
& | \ p; p & \text{sequential composition} \\
\multicolumn{3}{c}{\text{where } G \text{ may be a core gadget or a defined gadget.}}
\end{array}
$$

Fig. 5: A Small Language for Describing Circuits

control-flow, restricted table lookups and array manipulation. We detail practical implementation trivia in Section 7.

Importantly, our type system is split in two layers. The first layer simply distinguishes public variables (and their size, for example, using sized unsigned integer types) from sensitive shared variables in $\widetilde{\mathbb{K}}$. The second layer of the type system keeps track, for sensitive variables, of the subsets of their shares that are required to simulate the gadgets in which they are used. We focus here on this second layer, leaving details of the first layer to Section 7. Our compiler does not decide whether a particular input is secret or public, or whether a particular output is safe to declassify and unmask. Rather, we are content with propagating and making use of user-provided annotations on the inputs and outputs of a circuit through our type-checker. These annotations are part of the theorem statements and should be checked carefully by the user.

Formally, we see programs in our language as gadgets from states to states, where a state $\mathsf{St} \in \mathsf{Var} \to \widetilde{\mathbb{K}}$ simply maps each variable in the circuit (in $\mathsf{Var}$) to the value of its shares. To each program point (or gadget), we wish to associate a *concrete typemap* $\Gamma \in \mathsf{Var} \to [0..m)$ that intuitively keeps track, for each variable (or bundle), of a set of its shares that is sufficient to simulate the leakage of later gadgets in the circuit. However, computing the concrete typemaps for each possible set of adversary observations would quickly become unfeasible in practice, and we instead compute *symbolic typemaps over* $\mathbb{X}$, that is, $\Gamma \in \mathsf{Var} \to \mathcal{P}(\mathbb{X})$ where $\mathbb{O}$, $\mathbb{I}$ and $\mathbb{S}$ are disjoint sets of names. Intuitively, we will use symbols in $\mathbb{O}$ to denote adversary observations on a circuit's outputs, symbols in $\mathbb{I}$ to denote adversary observations on a gadget's intermediate variables (associating a symbol in $\mathbb{I}$ to each core gadget in the circuit), and symbols in $\mathbb{S}$

to denote existentially quantified sets of shares, as given to us by the non-interference properties of core gadgets. Given a set of abstract names $\mathbb{X}$ (in $\mathcal{P}(\mathbb{O} \cup \mathbb{I} \cup \mathbb{S})$), we use $[\![\mathbb{X}]\!]$ to denote the *interpretation* of $\mathbb{X}$, that is the set of concrete sets of indices that can be used to instantiate them. We use standard notations for map lookups ($\Gamma[\cdot]$) and updates ($\Gamma[\cdot \mapsto \cdot]$). We extend the notion of input and output projections indexed by sets of shares to state projections indexed by typemaps as follows.

Given a state $\mathsf{St} \in \mathsf{Var} \to \widetilde{\mathbb{K}}$ and a concrete typemap $\Gamma \in \mathsf{Var} \to [0..m)$, we define the *projected state* $\pi_\Gamma \circ \mathsf{St}$ as the map $\pi_\Gamma \circ \mathsf{St}\ x = \pi_{\bigcup_{X \in \Gamma[x]} X}\ \mathsf{St}[x]$, that is, the reduced state where each $x$ only maps to those shares that appear in its type by $\Gamma$.

### 5.2 $t$-NI by Typing

We consider typing judgments of the form

$$\vdash_{\mathbb{O},\mathbb{I} \uplus \mathbb{I}^p,\mathbb{S} \uplus \mathbb{S}'} \{C_{\mathsf{pre}}; \Gamma_{\mathsf{pre}}\}\ p\ \{C_{\mathsf{post}}; \Gamma_{\mathsf{post}}\}$$

where $C_{\mathsf{pre}}$ and $\Gamma_{\mathsf{pre}}$ are a constraint and a typemap over $\mathbb{O} \cup \mathbb{I} \cup \mathbb{I}^p \cup \mathbb{S} \cup \mathbb{S}'$, and $C_{\mathsf{post}}$ and $\Gamma_{\mathsf{post}}$ are a constraint and a typemap over $\mathbb{O} \cup \mathbb{I} \cup \mathbb{S}$. We say that such a judgment is *valid* whenever, for all $\mathcal{O} \in [\![\mathbb{O}]\!]$, for all $\mathcal{I} \in [\![\mathbb{I}]\!]$, for all $\mathcal{I}^p \in [\![\mathbb{I}^p]\!]$, and for all $\mathcal{S} \in [\![\mathbb{S}]\!]$ such that $C_{\mathsf{post}} \wedge |\mathcal{O}| + |\mathcal{I}| \leq t$, there exists $\mathcal{S}' \in [\![\mathbb{S}']\!]$ such that $p$, seen as a gadget from state to state, is $(\Gamma_{\mathsf{pre}}, (\mathcal{I}^p, \Gamma_{\mathsf{post}}))$-NI.

Intuitively, $C_{\mathsf{post}}$ makes explicit the size constraints that are known to hold on symbols in $\mathbb{S}$ following the construction of the simulator for such observations (and also keeps track of the global constraint that $(\mathbb{I}, \mathbb{O})$ is $t$-compatible. The judgment holds whenever it is sufficient to know shares $\Gamma_{\mathsf{pre}}$ before executing $p$ in order to simulate the adversary's observations internal to $p$ itself (those in $\mathbb{I}^p$) and the shares of the final state indicated by $\Gamma_{\mathsf{post}}$. The construction of the simulator for $p$ produces sets of shares $\mathbb{S}'$ on $p$'s initial state and additional constraints given in $C_{\mathsf{pre}}$. During the course of type-checking, we will always construct judgments such that the shares of the final state indicated by $\Gamma_{\mathsf{post}}$ are sufficient to simulate the observations in $\mathbb{O}$ and $\mathbb{I}$.

To illustrate the expressiveness of this typing judgment, we use it to express the properties of some of the core gadgets discussed in Section 3 (as well as an affine gadget). We write arg and res for the input and output bundles of each gadget (in some cases, arg may be a vector of bundles).

The Refresh gadget can be given the following type

$$\vdash_{\{\mathcal{O}\},\emptyset \uplus \{\mathcal{I}\},\emptyset \uplus \{\mathcal{S}\}} \{|\mathcal{S}| \leq |\mathcal{I}| \wedge C_0; \Gamma_0[\mathsf{arg} \mapsto \{\mathcal{S}\}]\}\ \mathsf{Refresh}\ \{C_0; \Gamma_0\}$$

where $C_0 = |\mathcal{O}| + |\mathcal{I}| \leq t$ is the global constraint on adversary observations and $\Gamma_0$ is the map that associates to res the set of names $\{\mathcal{O}\}$, and is empty everywhere else. This type states exactly the $t$-SNI property of Refresh: for all $(\mathcal{I}, \mathcal{O})$ such that $|\mathcal{I}| + |\mathcal{O}| \leq t$, there exists $\mathcal{S}$ such that $|\mathcal{S}| \leq |\mathcal{I}|$ and Refresh is $(\Gamma_0[\mathsf{arg} \mapsto \mathcal{S}], (\mathcal{I}, \Gamma_0))$-NI.

The $\odot$ gadget can be given the following type

$$\vdash_{\{\mathcal{O}\},\emptyset \uplus \{\mathcal{I}\},\emptyset \uplus \{\mathcal{S}_1,\mathcal{S}_2\}} \{|\mathcal{S}_1| \leq |\mathcal{I}| \wedge |\mathcal{S}_2| \leq |\mathcal{I}| \wedge C_0; \Gamma_0[\mathsf{arg}_i \mapsto \{\mathcal{S}_i\}]\}\ \odot\ \{C_0; \Gamma_0\}$$

where $C_0 = |\mathcal{O}| + |\mathcal{I}| \leq t$ is the global constraint on adversary observations and $\Gamma_0$ is the map that associates to res the set of names $\{\mathcal{O}\}$, and is empty everywhere else. This

type states exactly the $t$-SNI property of $\odot$: for all $(\mathcal{I}, \mathcal{O})$ such that $|\mathcal{I}| + |\mathcal{O}| \leq t$, there exist $\mathcal{S}_1, \mathcal{S}_2$ such that $|\mathcal{S}_i| \leq |\mathcal{I}|$ and $\odot$ is $(\Gamma_0[\mathsf{arg}_i \mapsto \mathcal{S}_i], (\mathcal{I}, \Gamma_0))$-NI.

The $\oplus$ gadget can be given the following type

$$\vdash_{\{\mathcal{O}\},\emptyset\uplus\{\mathcal{I}\},\emptyset\uplus\{\widehat{\mathcal{S}}\}} \{|\widehat{\mathcal{S}}| \leq |\mathcal{I}| \wedge C_0; \Gamma_0[\mathsf{arg}_i \mapsto \{\widehat{\mathcal{S}}, \mathcal{O}\}]\} \ \oplus \ \{C_0; \Gamma_0\}$$

where $C_0 = \mathsf{true}$ is the trivial global constraint on adversary observations and $\Gamma_0$ is the map that associates to $\mathsf{res}$ the set of names $\{\mathcal{O}\}$, and is empty everywhere else. This type is exactly the lifting to states of the affine-NI property of $\oplus$: for all $(\mathcal{I}, \mathcal{O})$, there exists $\widehat{\mathcal{S}}$ such that $|\widehat{\mathcal{S}}| \leq |\mathcal{I}|$ and $\oplus$ is $(\Gamma_0[\mathsf{arg}_i \mapsto \widehat{\mathcal{S}} \cup \mathcal{O}], (\mathcal{I}, \Gamma_0))$-NI.

In order to express our main theorem, we also consider arithmetic judgments of the form

$$\mathbb{O}; \mathbb{I}; \mathbb{S}; C \vDash_P \Gamma$$

where $P$ is a *whole circuit*, with designated input and output bundles. Such a judgment is said to be *valid* whenever for all $\mathcal{O} \in [\![\mathbb{O}]\!]$, for all $\mathcal{I} \in [\![\mathbb{I}]\!]$, and for all $\mathcal{S} \in [\![\mathbb{S}]\!]$ such that $C$ holds then for each variable $x$ that is an input to $P$, we have $|\bigcup_{X \in \Gamma[x]} X| \leq t$.

We now state our main composition theorem, Theorem 3, which states the security of well-typed programs under constraints on the size of the inferred types of its input variables.

**Theorem 3 (Non-Interference by Typing).** *Given a circuit $P$ if the following judgments are valid, then $P$ is $t$-NI.*

$$\vdash_{\mathbb{O},\emptyset\uplus\mathbb{I},\emptyset\uplus\mathbb{S}} \{C_{\mathsf{pre}}; \Gamma_{\mathsf{pre}}\} \ P \ \{C_0; \Gamma_0\} \tag{1}$$

$$\mathbb{O}; \mathbb{I}; \mathbb{S}; C_{\mathsf{pre}} \vDash_P \Gamma_{\mathsf{pre}} \tag{2}$$

*with $C_0 = |\mathbb{O}| \leq t$; and $\Gamma_0$ the typemap that associates, to each output bundle of $P$ a different symbol in $\mathbb{O}$.*

It is relatively easy to come up with a simple type system that formalizes and automates the kind of reasoning used in the proof of Theorem 2. However, we wish to make it more efficient by allowing sub-procedures, such as that computing the AES S-box, to be type-checked independently of context and used anywhere whilst retaining both soundness and a sufficient level of precision. This is rather difficult in general. For example, consider the gadget shown in Figure 6 (which is affine). It is perfectly safe to use it as is in the context of Figure 7 (which is also affine), but using it in the circuit shown in Figure 8 leads to an insecure algorithm. On the other hand, making it safe to use in any context (by refreshing, say, its first output) makes its use in Figure 7 less efficient than necessary.

We describe our type system, in all its generality, in Appendix D.

## 6 Stronger Composition Results

So far, we have shown that a simple compositional proof system and some machine-checked proofs are sufficient to prove an algorithm secure in Ishai, Sahai and Wagner's

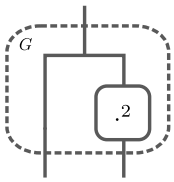



Fig. 6: A gadget G

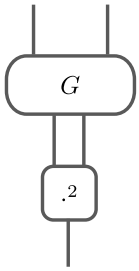



Fig. 7: A safe use of G

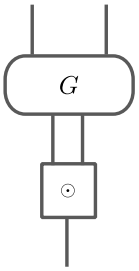



Fig. 8: An unsafe use of G

*stateless* $t$-threshold probing model [15]. However, this is not enough to guarantee security against an adversary that may move probes between oracle queries when these queries make use of some shared secret state. For instance, if an adversary can move his probes between two executions of an AES (with related inputs/outputs), then the aforementioned compositional properties do not hold anymore. To protect against such an adversary, the literature [15,8,9] recommends to protect the circuit using $2t + 1$ shares and refresh the entirety of the secret state between oracle queries.

We propose a novel method to protect an algorithm against adaptive probing that does not require doubling the number of shares in the entire circuit. Rather, we only double the number of shares on the state when it is stored between oracle queries. More clearly, the state is stored as $2m = 2t + 2$ shares, but computation is performed over $m = t + 1$ shares only. To enable this, we rely on algorithms Double and Half (Algorithm 6), that double the number of shares and divide it by 2, respectively, in combination with a mask refreshing gadget over $2m$ shares. The security proof for this mechanism is made feasible by the compositional proof system presented in this paper.

**Algorithm 6** Robust Mask Refreshing: Double and Half

**function** Double($\boldsymbol{a}$)
  **for** $i = 0$ **to** $t$ **do**
    $c_{2i} \xleftarrow{\$} \mathbb{K}$
    $c_{2i+1} \leftarrow a_i \ominus c_{2i}$
  **return** $\boldsymbol{c}$

**function** Half($\boldsymbol{a}$)
  **for** $i = 0$ **to** $t$ **do**
    $c_i \leftarrow a_{2i} \oplus a_{2i+1}$
  **return** $\boldsymbol{c}$

For simplicity, we express our composition theorem (Theorem 4) on gadgets that have a single input and a single output, that we use to model the state. It is easy to generalize to arbitrary scenarios, taking care to use Double, Refresh and Half as specified on *all* the variables that encode the shared state in both gadgets.

**Theorem 4 (Robust Composition).** *Given two $t$-TNI gadgets $F$ and $G$, for any $t$-admissible observations set $\Omega^{F'}$ on $F'$ and any $t$-admissible observation set $\Omega^{G'}$ on $G'$, there exists a $|\Omega^{F'}|$-compatible $\mathcal{S}$ such that the composition $F'; G'$ described below is $(\mathcal{S}, (\Omega^{F'}, \Omega^{G'}))$-NI.*

$$\left.\begin{array}{l} y \leftarrow F(x); \\ \bar{y} \leftarrow \mathsf{Double}(y); \\ \bar{y} \leftarrow \mathsf{Refresh}_{2t+2}(\bar{y}); \end{array}\right\} F' \qquad \left.\begin{array}{l} \bar{y} \leftarrow \mathsf{Refresh}_{2t+2}(\bar{y}); \\ y \leftarrow \mathsf{Half}(\bar{y}); \\ z \leftarrow G(y); \end{array}\right\} G'$$

Intuitively, any set made of $t_1 \leqslant t$ observations on $F'$ and $t_2 \leqslant t$ observations on $G'$ can be perfectly simulated by only $t_1$ shares of each $F'$'s input.

**Corollary 1.** *Any number $n$ of $t$-TNI gadgets $(G_i)_{0 \leq i < n}$ can be composed in a robust way as outlined in Theorem 4.*

The proof of Theorem 4 is given in Appendix C, which also discusses how these results can be used to prove that a masked algorithm is secure in the stateful and adaptive probing model of Ishai, Sahai and Wagner [15].

## 7 Implementation and Evaluation

As a proof of concept, we implement our compiler to read and produce programs in a reasonable subset of C (including basic operators, constant for loops, table lookups at *public* indices in public tables, and mutable secret state), equipped with libraries implementing core and extended operations for some choices of $\mathbb{K}$. The techniques and results we present in this paper are in no way restricted to this language and could be adapted to many other settings (ASM or VHDL, for example) given a concrete target. We see such an adaptation purely as a programming language challenge, as it requires properly formalizing the semantics and side-channels of such low-level platforms. As is standard in compilation, our compiler performs several passes over the code in order to produce its final result.

### 7.1 Compiler Implementation

**Parsing and Pre-Typing.** This pass parses C code into our internal representation, checks that the program is within the supported subset of C, performs C type-checking and checks that variables marked as sensitive (variables given type $\mathbb{K}$) are never implicitly cast to public types. Implicit casts from public types to $\mathbb{K}$ (when compatible, for example, when casting a public `uint8_t` to a protected variable in $\mathsf{GF}(2^8)$) are replaced with public sharing gadgets.

**Gadget Selection and Generic Optimizations.** This pass heuristically selects optimal gadgets depending on their usage. For example, multiplication of a secret by a public value can be computed by an affine gadget that multiplies each share of the secret, whereas the multiplication of two secrets must be performed using the SecMult gadget. Further efforts in formally proving precise types for specialized core gadgets may also improve this optimization step. This pass also transforms the C code to clarify calling conventions (ensuring that arguments are passed by value when necessary), to make it follow a simpler form (see Figure 5) that makes it easier to type-check, and to optimize the use of intermediate registers.

**Type-Checking and Refresh Insertion.** This is the core of our compiler. We note that the type system from Section 5 fails exactly when a mask refreshing operation is needed. At the cost of tracking some more information and reinforcing the typing constraint on sub-gadgets, we use this observation to automatically insert Refresh gadgets where required. When type-checking fails, the variable whose masks need to be refreshed is duplicated and one of its uses is replaced with the refreshed duplicate. To avoid having to re-type the entire program after insertion of a refresh gadget, our compiler keeps track of typing information for each program point already traversed and simply rewinds the typing to the program point immediately after the last modification.

The source program can also be annotated with explicit refresh operations that may help the compiler in its type-checking operations. The compiler itself can also be run in a mode that only performs type-checking and reports failures without attempting to correct the program, allowing it to be used for the direct verification of implementations clearly structured as compositions of gadgets.

**Code-Generation.** Finally, once all necessary mask refreshing operations have been inserted and the program has been type-checked, we produce a masked C program. This transformation is almost a one-to-one mapping from the instructions in the type-checked programs to calls to a library of verified core gadgets or to newly defined gadgets. Some cleanup is performed on loops to clarify the final code whenever possible.

### 7.2 Practical Evaluation

To test the effectiveness of our compiler, we apply it to implementations of different algorithms, generating masked implementations at various orders. We apply our compiler to the following programs: **AES ($\odot$),** a full block of AES-128 masked using the standard multiplication gadget, and implemented in $\mathsf{GF}(2^8)$ (see Appendix E for input and output S-box algorithms); **AES ($x \odot g(x)$),** a full block of AES-128 masked using Coron et al.'s gadget for computing $x \odot g(x)$, and implemented in $\mathsf{GF}(2^8)$; **Keccak,** a full computation (24 rounds) of Keccak-f[1600], implemented in $\mathbb{B}^{64}$; **Simon,** a block of Simon(128,128), implemented in $\mathsf{GF}(2^{64})$; **Speck,** a block of Speck(128,128), implemented in $\mathbb{B}^{64}$.

Speck makes use of both bitwise operations (that are difficult to perform on additively shared variables) and modular addition (which is costly to perform on boolean shared variable). We choose to mask Speck using a Boolean secret-sharing scheme, implementing the algorithm in $\mathbb{B}^{64}$ and using Coron, Großschädl and Vadnala's algorithm to compute modular addition directly on boolean-shared variables [7]. Since this is a defined gadget, it is compiled as part of the program and the security of its masked version is proved by typing during compilation.

Table 1 shows resource usage statistics for generating the masked algorithms (at any order) from unmasked implementations of each algorithm. The table shows the total number of mask refreshing operations inserted in the program[8], the compilation

[8] Note that the number of mask refreshing operations executed during an execution of the program may be much greater, since the sub-procedure in which the insertion occurs may be called multiple times.

time, and the memory consumption. The first Keccak line refers to an implementation of Keccak where the mask refreshing operation is already inserted (as a noop) in the unmasked algorithm, and the compiler is run in pure verification mode. The second line refers to the compiler being run on a purely unmasked algorithm. The difficulty here comes from the large number of mask refreshing gadgets that need to be inserted, requiring the type-checker to backtrack and start over multiple times. However, first running the type-checker on a round reduced version of Keccak and identifying the problematic program point allows the user to manually insert the mask refreshing operation and simply use the compiler as a verifier to check that the resulting algorithm is indeed secure. Apart from this extreme example, which is due to the particular shape of the Keccak $\chi$ function and the way it is used in Keccak-f's round function, all compilations are rather cheap. Also note that even costly compilations only have to be performed *once* to transform an unmasked algorithm into a masked algorithm that is secure at any order.

Table 1: Time taken to generate masked implementation at any order

| Scheme | # Refresh | Time | Memory |
|---|---|---|---|
| AES ($\odot$) | 2 | 0.09s | 4Mo |
| AES ($x \odot g(x)$) | 0 | 0.05s | 4Mo |
| Keccak | 0 | 121.20 | 456Mo |
| Keccak(2) | 600 | 2728.00s | 22870Mo |
| Simon | 67 | 0.38s | 15Mo |
| Speck | 61 | 6.22s | 38Mo |

*Remark 2.* The compiler reports that the modular addition gadget used in Speck is indeed secure without inserting any refresh gadgets. This serves as a compositional and machine-checked proof of security for this algorithm at any order $t$.

Table 2 reports the time taken to execute the resulting programs 10,000 times at various orders.[9]

Table 2: Time taken by 10,000 executions of each program at various masking orders

| Scheme | unmasked | Order 1 | Order 2 | Order 3 | Order 5 | Order 10 | Order 15 | Order 20 |
|---|---|---|---|---|---|---|---|---|
| AES ($\odot$) | 0.078s | 2.697s | 3.326s | 4.516s | 8.161s | 21.318s | 38.007s | 59.567s |
| AES ($x \otimes g(x)$) | 0.078s | 2.278s | 3.209s | 4.368s | 7.707s | 17.875s | 32.552s | 50.588s |
| Keccak | 0.238s | 1.572s | 3.057s | 5.801s | 13.505s | 42.764s | 92.476s | 156.050s |
| Simon | 0.053s | 0.279s | 0.526s | 0.873s | 1.782s | 6.136s | 11.551s | 20.140s |
| Speck | 0.022s | 4.361s | 10.281s | 20.053s | 47.389s | 231.423s | 357.153s | 603.261s |

[9] On a Intel(R) Xeon(R) CPU E5-2667 0 @ 2.90GHz with 64Go of memory running Linux (Fedora)

For AES and Speck, the figures shown in the "unmasked" column are execution times for the input to our compiler: a table-based implementation of AES or an implementation of Speck that uses machine arithmetic, rather than Coron, Großschädl and Vadnala's algorithm would be much faster, but cannot be masked directly. As an additional test to assess the performance of generated algorithms at very high order, we ran 10,000 instances of AES masked at order 100; it took less than 18 minutes to complete (so 0.108 secondes per instance).

Modifying our compiler to make use of the addition-based mask refreshing gadget shown in Algorithm 2 (and therefore produce insecure programs) yields virtually the same figures as those shown for AES, highlighting the fact that the cost of using a secure mask refreshing gadget is negligible. However, it is important to keep in mind that fresh randomness is much less costly in our setting than, say, on an embedded device, and that the use of a secure refresh gadget may cause a more important performance degradation in such a setting.

## 8 Conclusion

We have addressed the theoretical problem of secure composition in higher-order masking implementations, by introducing the notion of strong simulatability, and by showing that it supports provably correct compositional security analyses. We have displayed a general method to verify composition or to build a higher-order secure algorithm by properly positioning strong simulatable refresh gadgets. Moreover, we have observed that other gadgets from the literature are strongly simulatable, thereby reducing the needs in refresh gadgets instances, and leading to more efficient design. To exploit these new results on large circuits, we have constructed a concrete and efficient compiler, which allows us to obtain secure implementations for masking at any higher-order of several cryptographic algorithms. Finally, we have extended our compilation results to protect protocols during which the adversary is able to regularly move his probes. There are many avenues to extend our theoretical results and program transformation. Given that the masked algorithms generated by our tool are provably secure in the $t$-threshold probing model, and that the relationship of this model with the noisy leakage model, we expect that our algorithms will resist practical attacks, but it would also be comforting to carry out a practical security evaluation of our algorithms and validate their security experimentally.

*Acknowledgments* Pierre-Yves Strub contributed greatly to early discussions and to the compiler implementation and infrastructure. Discussions with Thomas Roche were useful in developing the pen and paper proofs of $t$-SNI for core gadgets, and in particular for the secure multiplication algorithm.

This work was partially supported by Spanish project S2013/ICE-2731 N-GREENS Software-CM and TIN2012-39391-C04-01 StrongSoft, Madrid regional project S2013/ICE-2731 N-GREENS Software-CM, and ANR projects ANR-10-SEGI-015 PRINCE and ANR-14-CE28-0015 BRUTUS. The third author's research is supported by an FP7 Marie Curie Actions-COFUND programme (grant no. 291803).

## References


1. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, April 2015.
2. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial, 2013.
3. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, August 2011.
4. Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 293–310. Springer, August 2013.
5. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, August 1999.
6. Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, May 2014.
7. Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, September 2014.
8. Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, March 2014.
9. Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, May 2014.
10. Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 401–429. Springer, April 2015.
11. Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification, CAV 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014.
12. Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2014.
13. Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from computationally bounded and noisy leakage. *SIAM Journal on Computing*, 43(5):1564–1614, 2014.
14. Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Thwarting higher-order side channel analysis with additive and multiplicative maskings. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 240–255. Springer, September / October 2011.

15. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, August 2003.
16. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, August 1996.
17. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, August 1999.
18. Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 278–296. Springer, February 2004.
19. Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 58–75. Springer, September 2012.
20. Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, May 2013.
21. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, August 2010.
22. François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 443–461. Springer, April 2009.
23. François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order DPA. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 112–129. Springer, December 2010.

## A Proof of Lemma 1

*Proof (Lemma 1).* We consider a $(n, \ell, o)$-gadget $G$, a compatible $\mathcal{S}$ and an admissible $\Omega$.

We first prove that $(\mathcal{S}, \Omega)$-SIM implies $(\mathcal{S}, \Omega)$-NI. Assume that $G$ is $(\mathcal{S}, \Omega)$-SIM. For any $\boldsymbol{s}_0$, $\boldsymbol{s}_1$ in $\widetilde{\mathbb{K}}^n$ such that $\pi_{\mathcal{S}}(\boldsymbol{s}_0) = \pi_{\mathcal{S}}(\boldsymbol{s}_1)$, we have $\pi_\Omega \circ G(\boldsymbol{s}_0) \overset{Def.\ 2}{=} G_\Omega \circ \pi_{\mathcal{S}}(\boldsymbol{s}_0) = G_\Omega \circ \pi_{\mathcal{S}}(\boldsymbol{s}_1) \overset{Def.\ 2}{=} \pi_\Omega \circ G(\boldsymbol{s}_1)$. Thus, $G$ is $(\mathcal{S}, \Omega)$-NI.

Assume now that $G$ is $(\mathcal{S}, \Omega)$-NI. From Definition 2, we have $\pi_\Omega \circ G(\boldsymbol{s}_0) = \pi_\Omega \circ G(\boldsymbol{s}_1)$ for any $\boldsymbol{s}_0$, $\boldsymbol{s}_1$ in $\widetilde{\mathbb{K}}^n$ such that $\pi_{\mathcal{S}}(\boldsymbol{s}_0) = \pi_{\mathcal{S}}(\boldsymbol{s}_1)$. We construct a simulator $G_\Omega$ by returning, given a projected input $\boldsymbol{\sigma}$, the result of $\pi_\Omega \circ G(\boldsymbol{\sigma}||0)$ for wires in $\Omega$ and random values on all other wires (where $\boldsymbol{\sigma}||0$ injects $\boldsymbol{\sigma}$ into $\widetilde{\mathbb{K}}^n$ by assigning 0 to all missing inputs). Since $\pi_{\mathcal{S}}(\pi_{\mathcal{S}}(\boldsymbol{s})||0) = \pi_S(\boldsymbol{s})$ for all full input $\boldsymbol{s}$, we have $G_\Omega \circ \pi_{\mathcal{S}}(\boldsymbol{s}) = \pi_\Omega \circ G(\pi_{\mathcal{S}}(\boldsymbol{s})||0) \overset{Def.2}{=} \pi_\Omega \circ G(\boldsymbol{s})$ for all full input $\boldsymbol{s}$. Thus, $G$ is $(\mathcal{S}, \Omega)$-SIM.

## B Proofs of Strong Simulatability

We prove that the gadgets we consider are $t$-SNI and functionally correct. The definition of correctness is standard and can be stated formally using the notion of unmasking, where for any $n$ and $x \in \mathbb{K}^n$, the *unmasking of* $x$ is defined as the sum $[\![x]\!] = \bigoplus_{i=0}^{n-1} x_i$.

### B.1 Multiplication-Based Refreshing

We provide a pen-and-paper proof of Proposition 1; the proof matches closely its formalization in EasyCrypt.

*Proof (Proposition 1).* For functional correctness, we have to prove that the algorithm implements the identity function: $[\![\mathsf{RefreshMult}(a)]\!] = [\![a]\!]$. This can be seen by expanding its results and simplifying the sums.

To prove $t$-SNI, we construct a simulator similar to those previously used to prove the $t$-NI of several masking transformations (e.g., [15,21,8]). Let $\Omega = (\mathcal{I}, \mathcal{O})$ be a $t$-admissible set of observations, and let $d_1 = |\mathcal{I}|$ and $d_2 = |\mathcal{O}|$. Note that $d_1 + d_2 \leq t$. Our goals are: i. to find a $d_1$-compatible set $\mathcal{S}$, ii. to construct a perfect simulator that uses only shares $\mathcal{S}$ of the inputs.

First, we identify which variables are internals and which are outputs. Internals are the $a_i$, the $r_{i,j}$ (the value of $r$ at iteration $i, j$), and the $c_{i,j}$ (resp. $c_{j,i}$) which correspond to the value of the variable $c_i$ (resp. $c_j$) at iteration $i, j$. Outputs are the final values of $c_i$ (i.e. $c_{i,t}$).

We define $\mathcal{S}$ as follows: for each observation among $a_i$, $r_{i,j}$ and $c_{i,j}$ (with $j < t$) we add the index $i$ to $\mathcal{S}$. It is clear that $\mathcal{S}$ contains at most $d_1$ indices. We now construct the simulator. For clarity, observe that the RefreshMult algorithm can be equivalently

represented using the following matrix:

$$\begin{pmatrix} a_0 & 0 & r_{0,1} & r_{0,2} & \cdots & r_{0,t} \\ a_1 & \ominus\, r_{0,1} & 0 & r_{1,2} & \cdots & r_{1,t} \\ a_2 & \ominus r_{0,2} & \ominus r_{1,2} & 0 & \cdots & r_{2,t} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_d & \ominus\, r_{0,t} & \ominus\, r_{1,t} & \ominus r_{2,t} & \cdots & 0 \end{pmatrix}.$$

In this setting, $c_{i,j}$ corresponds to the partial sum of the $j+2$ first elements of line $i$. For each $i \in \mathcal{S}$ (that is, for each line $i$ that contains at least one observed internal value), $a_i$ is provided to the simulator (by definition of $\mathcal{S}$). Thus, the simulator can sample all $r_{i,j}$ and compute all partial sums $c_{i,j}$ and the $i$th output normally. At this point, all values on line $i \in \mathcal{S}$ (internal and output) are perfectly simulated.

We still have to simulate the observed output values for rows on which no internal values are observed. Remark that simulating the $i$th line also necessarily fixes the value of all random variables appearing in the $i$th column (so that dependencies between variables are preserved). After internal observations are simulated, at most $d_1$ lines of the matrix are fully filled. Therefore, at least $t - d_1 \geq d_2$ random values are unsimulated on lines on which no internal observations are made. For each output observation made on one such line (say $i$), we can therefore pick a different $r_{i,j}$ that we fix so that output $i$ can be simulated using a freshly sampled uniform value. □

In order to formally verify this proof with EasyCrypt, we need to prove the equivalence between two programs which share the same inputs $\{a_i\}_{i \in I}$. That is, we show that whatever the observations made by adversary, as soon as they are upper bounded by $t$, he cannot distinguish between both programs. To proceed, we need to write different derivations of the original program in order to simplify the proof of equivalence for Easycrypt. Thus, we organize the security proof of Proposition 1 as a sequence of games and a sequence of codes. We use different colors in the codes to underline the differences with the previous game.

**Game 0.** The first game represents the original refreshing function RefreshMult which computes the $t+1$ shares $c_i$ without any observation.

```
function RefreshMult(a) :
for i = 0 to t do
  c_i ← a_i;
for i = 0 to t do
  for j = i + 1 to t
    r_{i,j} ← $;
    c_i ← c_i ⊕ r_{i,j};
    c_j ← c_j ⊖ r_{i,j};
return c
```

**Game 1.** The second game also represents the original refreshing function but on which the attacker can make up to $t$ observations whose $t_1$ on the internal variables and $t_2 = t - t_1$ on the outputs. All the observations are given in argument of the function and for each intermediate variable, if it belongs to the observations, it is stored in a table. We refer to this function as $R_0$.

As explained in Section 3, there are three kinds of internal observations $a_i$, $r_{i,j}$, $c_{i,j}$ and the final $c_i$. With this first function, we can build the subscript $I$. If we define by $\mathsf{ind_k}$ the $k^{th}$ index of a variable, we have $I = \mathsf{ind}_1(a_i) \cup \mathsf{ind}_1(r_{i,j}) \cup \mathsf{ind}_1(c_{i,j})$. In this step, we formally verify with Easycrypt that the cardinal of $I$ is at most $t_1$ and we show that RefreshMult and $R_0$ computes the same outputs.

**Function** $\mathsf{R}_0(\boldsymbol{a}, \boldsymbol{O})$ :
**for** $i = 0$ **to** $d$ **do**
  $c_i \leftarrow a_i$;
  **if** $(a_i \in O)$ **then** $\overline{a_i} \leftarrow a_i$;
**for** $i = 0$ **to** $d$ **do**
  **for** $j = i + 1$ **to** $d$ **do**
    $r_{i,j} \leftarrow \$$;
    **if** $(r_{i,j} \in O)$ **then** $\overline{r_{i,j}} \leftarrow r_{i,j}$;
    $c_i \leftarrow c_i \oplus r_{i,j}$;
    **if** $(c_{i,j} \in O)$ **then** $\overline{c_{i,j}} \leftarrow c_i$;
    $c_j \leftarrow c_j \ominus r_{i,j}$;
    **if** $(c_{j,i} \in O)$ **then** $\overline{c_{j,i}} \leftarrow c_j$;
**for** $i = 0$ **to** $d$ **do**
  **if** $(c_i \in O)$ **then** $\overline{c_i} \leftarrow c_i$;
**return** $\boldsymbol{c}$

**Game 2.** We make a few changes on $R_0$:

- we generate all the fresh random values at the beginning of the function with the random oracle-like function Sample and we store them in a matrix,
- we compute all the values corresponding to the internal observations,
- we compute the outputs $(c_i)_{i \in I}$,
- we compute the outputs $(c_i)_{i \notin I}$.

We refer to this new function as $R_1$ and we use EasyCrypt to prove that $R_0$ and $R_1$ are equivalent if they share the same inputs $(a_i)_{i \in I}$.

**Function** $\mathsf{SumCij}(i, j)$ :
$s \leftarrow a_i$;
**for** $k = 0$ **to** $j$ **do**
  **if** $(i < k)$ **then** $s \leftarrow s \oplus \overline{r_{i,k}}$;
  **elseif** $(i > k)$ **then** $s \leftarrow s \ominus \overline{r_{k,i}}$;
**return** $s$;

**Function** $\mathsf{R}_1(\boldsymbol{a}, \boldsymbol{O})$ :
**for** $i = 0$ **to** $t$ **do**
  **for** $j = i + 1$ **to** $t$ **do**
    $\overline{r_{i,j}} \leftarrow \mathsf{Sample}(i, j)$;
**for** $i = 0$ **to** $t$ **do**
  **if** $(a_i \in O)$ **then** $\overline{a_i} \leftarrow a_i$;
**for** $i = 0$ **to** $t$ **do**
  **if** $(i \in I)$ **then**
    **for** $j = 0$ **to** $t$ **do**
      **if** $(r_{i,j} \in O)$ **then**
        $r_{i,j} \leftarrow \overline{r_{i,j}}$;
      **if** $(c_{i,j} \in O)$ **then**
        $c_{i,j} \leftarrow \mathsf{SumCij}(i, j)$;
    **if** $(c_i \in O)$ **then**
      $c_i \leftarrow \mathsf{SumCij}(i, t)$;
**for** $i = 0$ **to** $d$ **do**
  **if** $(i \notin I)$ **then**
    **if** $(c_i \in O)$ **then**
      $c_i \leftarrow \mathsf{SumCij}(i, t)$;
**return** $\boldsymbol{c}$

**Game 3.** We now make a conceptual change to Game 2. We delay the generation of the random values as late as possible, i.e., that is just before their first use. We refer to this new function as $R_2$. We prove the equivalence between Games 2 and 3 if they share the same inputs $\{a_i\}_{i\in I}$ with a generic proof made in EagerLazy.

**Function** $\mathsf{SumCij}(i,j)$ :
$s \leftarrow a_i$;
**for** $k = 0$ **to** $j$ **do**
 **if** $(i < k)$ **then** $s \leftarrow s \oplus \mathsf{Sample}(i,k)$;
 **elseif** $(i > k)$ **then** $s \leftarrow s \ominus \mathsf{Sample}(k,i)$;
**return** $s$;

**Function** $\mathsf{R}_2(\boldsymbol{a}, \boldsymbol{O})$ :
**for** $i = 0$ **to** $t$ **do**
 **if** $(a_i \in O)$ **then** $\overline{a_i} \leftarrow a_i$;
**for** $i = 0$ **to** $t$ **do**
 **if** $(i \in I)$ **then**
  **for** $j = 0$ **to** $d$ **do**
   **if** $(r_{i,j} \in O)$ **then**
    $r_{i,j} \leftarrow \mathsf{Sample}(i,j)$;
   **if** $(c_{i,j} \in O)$ **then**
    $c_{i,j} \leftarrow \mathsf{SumCij}(i,j)$;
  **if** $(c_i \in O)$ **then**
   $c_i \leftarrow \mathsf{SumCij}(i,t)$;
**for** $i = 0$ **to** $d$ **do**
 **if** $(i \notin I)$ **then**
  **if** $(c_i \in O)$ **then**
   $c_i \leftarrow \mathsf{SumCij}(i,t)$;
**return** $\boldsymbol{c}$;

**Game 4.** In this game, we make a significant change in the computation of the $c_i$ for all the $i$ which are not in $I$. Concretely, we show that there exist a non empty set of indices $L$ such that $\forall \ell \in L,\ r_{i,\ell}$ is not assigned yet. Then, instead of computing the $c_i$ (for $i \notin I$) as follows:

$$\forall \ell \in L,\ r_{i,\ell} \leftarrow \$,$$
$$c_i \leftarrow a_i \oplus \bigoplus_{j=0}^{i-1} r_{i,j} \ominus \bigoplus_{j=i+1}^{t} r_{i,j},$$

we make the following change:

$$c_i \leftarrow \$,$$
$$\forall \ell \in L \backslash \{k\},\ r_{i,\ell} \leftarrow \$,$$
$$\text{if } (i < k),\ \ r_{i,k} \leftarrow c_i \ominus a_i \ominus \bigoplus_{j=0, j\neq k}^{i-1} r_{i,j} \oplus \bigoplus_{j=i+1, j\neq k}^{t} r_{i,j},$$
$$\text{if } (i > k),\ \ r_{i,k} \leftarrow \ominus c_i \oplus a_i \oplus \bigoplus_{j=0, j\neq k}^{i-1} r_{i,j} \ominus \bigoplus_{j=i+1, j\neq k}^{t} r_{i,j}.$$

We prove the equivalence between Game 3 and Game 4 with EasyCrypt when functions $R_2$ and $R_3$ share the same inputs $\{a_i\}_{i\in I}$. The most critical part of this step is undoubtedly to ensure that the subscript $J$ contains at least one index. To do so, we need to show that the elements $r_{i,\ell}$ with $\ell \in L$ were not already used and won't be reused anywhere. Eventually, we formally prove that the results of $R_3$, which represents the final simulator, only depends on the inputs $\{a_i\}_{i\in I}$.

**Function** $\mathsf{SumCij}(i,j)$ :
$s \leftarrow a_i$;
**for** $k = 0$ **to** $j$ **do**
 **if** $(i < k)$ **then**
  $s \leftarrow s \oplus \mathsf{Sample}(i,k)$;
 **else if** $(i > k)$ **then**
  $s \leftarrow s \ominus \mathsf{Sample}(k,i)$;
**return** $s$;

**Function** $\mathsf{SetCi}(i)$;
$s \leftarrow a_i$; $k \leftarrow 0$;
**while** $((k \leqslant t) \wedge ((i == k) \vee$
 $(k < i \wedge ((i,k) \in \mathrm{dom}\ r)) \vee$
 $(i < k \wedge ((k,i) \in \mathrm{dom}\ r)))$ **do**
 **if** $(i < k)$ **then** $s \leftarrow s \oplus r_{i,k}$;
 **else then** $s \leftarrow s \ominus r_{i,k}$;
 $k \leftarrow k + 1$;
$k' \leftarrow k$; $r' \leftarrow \$$;
**for** $k = k'$ **to** $t$ **do**
 $s \leftarrow s \oplus r_{i,k}$;
**if** $(i < k')$ **then** $r_{i,k'} \leftarrow s \oplus r'$;
**else** $r_{k',i} \leftarrow s \ominus r'$;
**return** $r'$;

**Function** $\mathsf{R_3}(\boldsymbol{a}, \boldsymbol{O})$ :
**for** $i = 0$ **to** $t$ **do**
 **if** $(a_i \in O)$ **then**
  $\overline{a_i} \leftarrow a_i$;
**for** $i = 0$ **to** $t$ **do**
 **if** $(i \in I)$ **then**
  **for** $j = 0$ **to** $t$ **do**
   **if** $(r_{i,j} \in O)$ **then**
    $r_{i,j} \leftarrow \mathsf{Sample}(i,j)$;
   **if** $(c_{i,j} \in O)$ **then**
    $c_{i,j} \leftarrow \mathsf{SumCij}(i,j)$;
  **if** $(c_i \in O)$ **then**
   $c_i \leftarrow \mathsf{SetCi}(i)$;
**for** $i = 0$ **to** $t$ **do**
 **if** $(i \notin I)$ **then**
  **if** $(c_i \in O)$ **then**
   $c_i \leftarrow \mathsf{SetCi}(i)$;
**return** $\boldsymbol{c}$;

Finally, we have formally proved that an adversary could not distinguish between two programs which share the same inputs $\{a_i\}_{i \in I}$ with at most $t$ observations and that the cardinal of $I$ was upper bounded by the number of internal observations $d_1$.

### B.2 Secure Multiplication

We now prove Proposition 2 informally. A formal proof has also been done in EasyCrypt on the same model than for the multiplication-based refresh algorithm.

*Proof.* As for the multiplication-based mask refreshing algorithm, we first focus on the functional correctness by proving that the algorithm implements the field multiplication function: $[\![\mathsf{SecMult}(a,b)]\!] = [\![a \otimes b]\!]$. Similarly, this can be easily verified by simplifying the sums and by the ring axioms (in particular distributivity of $\otimes$ over $\oplus$).

Let $\Omega = (\mathcal{I}, \mathcal{O})$ be a $t$-admissible set of observations, and let $d_1 = |\mathcal{I}|$ and $d_2 = |\mathcal{O}|$. Note that $d_1 + d_2 \leq t$. To prove $t$-SNI, we aim to find two $d_1$-compatible sets $\mathcal{S}$ and $\mathcal{S}'$, and to construct a perfect simulator that uses only shares $\mathcal{S}$ of $\boldsymbol{a}$ and shares $\mathcal{S}'$ of $\boldsymbol{b}$.

First, we identify which variables are internals and which are outputs. We directly split the internals in four groups for the needs of the proof:

*Group 1:* the $a_i$, the $b_i$, and the $a_i \otimes b_i$,
*Group 2:* the $c_{i,j}$ (resp. $c_{j,i}$) which correspond to the value of the variable $c_i$ (resp. $c_j$) at iteration $i, j$,

*Group 3:* the $r_{i,j}$ (the first value of $r$ at iteration $i, j$), and the $r_{j,i}$ (the second value of $r$ at iteration $i, j$),
*Group 4:* the $a_i \otimes b_j$ and the $a_i \otimes b_j \ominus r_{i,j}$.

The output variables are the final values of $c_i$ (i.e., $c_{i,t}$).

As the algorithm takes two inputs $a$ and $b$, we define two subscripts $\mathcal{S}_a$ and $\mathcal{S}_b$ which will contain the indices of each input's shares that will be further used for the simulation of the observations. For each observation in the first or the second group, we add the index $i$ to $\mathcal{S}_a$ and to $\mathcal{S}_b$. For each observation in the third or in the fourth group: if the index $i$ is already in $\mathcal{S}_a$, we add the index $j$ to $\mathcal{S}_a$, otherwise we add the index $i$ to $\mathcal{S}_a$ and if the index $i$ is already in $\mathcal{S}_b$, we add the index $j$ to $\mathcal{S}_b$, otherwise we add the index $i$ to $\mathcal{S}_b$. It is clear that the final sets $\mathcal{S}_a$ and $\mathcal{S}_b$ contain each one at most $d_1$ indices, with $d_1 \leq t$.

We now construct the simulator. For clarity, observe that the SecMult algorithm can be equivalently represented using the following matrix:

$$\begin{pmatrix} a_0 \otimes b_0 & 0 & r_{0,1} & r_{0,2} & \cdots & r_{0,t} \\ a_1 \otimes b_1 & r_{1,0} & 0 & r_{1,2} & \cdots & r_{1,t} \\ a_2 \otimes b_2 & r_{2,0} & r_{2,1} & 0 & \cdots & r_{2,t} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_t \otimes b_t & r_{t,0} & r_{t,1} & r_{t,2} & \cdots & 0 \end{pmatrix}.$$

In this setting, $c_{i,j}$ corresponds to the partial sum of the $j+2$ first elements of line $i$. For each variable $r_{i,j}$ $(i < j)$ entering in the computation of an observation, we assign it a fresh random value. Then, for each observation in the first group, $a_i$ and $b_i$ are provided to the simulator (by definition of $\mathcal{S}_a$ and $\mathcal{S}_b$) thus the observation is perfectly simulated. For an observation in the third group, we distinguish two cases. If $i < j$, $r_{i,j}$ is already assigned to a fresh random value. If $i > j$, either $(i, j) \in \mathcal{S}_a \wedge \mathcal{S}_b$ and the observation is perfectly simulated from $r_{j,i}, a_i, b_i, a_j$ and $b_j$ or $r_{j,i}$ does not enter in the computation of any internal variable that was observed and $r_{i,j}$ (line 8) is assigned to a fresh random value. Each observation made in the fourth group is perfectly simulated using $r_{i,j}$, $a_i$ and $b_j$. As for an observation in the second group, the corresponding variable is a partial sum composed of a product $a_i \otimes b_i$ and of variables $r_{i,j}$. Since $a_i$ and $b_i$ are provided to the simulator in this case, we focus on each remaining $r_{i,j}$. Each one of them such that $i < j$ is already assigned to a fresh random value. For the others, if $r_{j,i}$ enters in the computation of any other internal observation, then $(i, j) \in \mathcal{S}_a \wedge \mathcal{S}_b$ and $r_{i,j}$ is simulated with $r_{j,i}, a_i, b_i, a_j$ and $b_j$. Otherwise, $r_{i,j}$ is assigned to a fresh random value.

We still have to simulate the observations on output variables. We start with the ones whose intermediate sums (group 2) are also observed. For each such variable $c_i$, the biggest partial sum which is observed is already simulated. Thus, we consider the remaining terms $r_{i,j}$. Each one of them such that $i < j$ is already assigned to a fresh random value. For the others, either $(i, j) \in (\mathcal{S}_a \cap \mathcal{S}_b)$ and $c_i$ is perfectly simulated from $r_{j,i}$, $a_i$, $b_i$, $a_j$ and $b_j$ or $r_{j,i}$ does not enter in the computation of any internal variable observed and $c_i$ is assigned to a fresh random value. We now consider output observations whose partial sums are not observed. Each of them is composed of $t$ $r_{i,j}$. And at most one of them can enter in the computation of each other variable $c_i$. Since,

we already considered (without this one) at most $t-1$ observations, at least one $r_{i,j}$ does not enter in the computation of any other observed variable. Thus, $c_i$ is assigned to a fresh random value.

### B.3 Gadget $x \otimes g(x)$

We provide a pen-and-paper proof of Proposition 3. This proof has not yet been formalized in EasyCrypt, and we leave this as future work if a fully certified compiler or verifier is desired.

*Proof (Proposition 3).* For functional correctness, we have to prove that the algorithm implements the function: $[\![h(x)]\!] = [\![x \otimes g(x)]\!]$. This can be seen by expanding its results and simplifying the field expressions.

Let $\Omega = (\mathcal{I}, \mathcal{O})$ be a $t$-admissible set of observations, and let $d_1 = |\mathcal{I}|$ and $d_2 = |\mathcal{O}|$. Note that $d_1 + d_2 \leq t$. To prove $t$-SNI, we need to: i. find a $d_1$-compatible set $\mathcal{S}$, ii. construct a perfect simulator that uses only shares $\mathcal{S}$ of the input.

First, we identify which variables are internals and which are outputs. For the sake of clarity, we denote by $r_{i,j}$ and $r'_{i,j}$ the random variables for which $i < j$. Internals are:

1. the $a_i$, the $g(a_i)$, the $a_i \otimes g(a_i)$ that only depend on $a_i$,
2. the $r'_{i,j}$, the $g(r'_{i,j})$, the $a_i \otimes g(r'_{i,j})$ and the $r'_{i,j} \otimes g(a_i)$ that depend on both $a_i$ and $r'_{i,j}$,
3. the $a_j \ominus r'_{i,j}$, the $g(a_j \ominus r'_{i,j})$, the $a_i \otimes g(a_j \ominus r'_{i,j})$ and the $(a_j \ominus r'_{i,j}) \otimes g(a_i)$ that depend on $a_i$, $a_j$ and $r'_{i,j}$,
4. the $r_{i,j}$, the $a_i \otimes g(r'_{i,j}) \ominus r_{i,j}$, the $a_i \otimes g(r'_{i,j}) \oplus r'_{i,j} \otimes g(a_i) \ominus r_{i,j}$, the $a_i \otimes g(r'_{i,j}) \oplus r'_{i,j} \otimes g(a_i) \oplus a_i \otimes g(a_j \ominus r'_{i,j}) \ominus r_{i,j}$ and the $a_i \otimes g(a_j) \oplus a_j \otimes g(a_i) \ominus r_{i,j}$ that are inversible in $r_{i,j}$,
5. the $a_i \otimes g(a_i) \oplus \bigoplus_{j=0}^{j_0} (a_i \otimes g(a_j) \oplus a_j \otimes g(a_i) \oplus r_{j,i})$ with $1 \leqslant j_0 \leqslant i-1$ and the $a_i \otimes g(a_i) \oplus \bigoplus_{j=0}^{i-1} (a_i \otimes g(a_j) \oplus a_j \otimes g(a_i) \ominus r_{j,i}) \oplus \bigoplus_{j=i+1}^{j_0} r_{i,j}$ with $i < j_0 < d$.

Outputs are the final values of $c_i$ (i.e. $c_{i,t}$).

We define $\mathcal{S}$ as follows. For each observation among the first or the fifth group we add the index $i$ to $\mathcal{S}$. For each observation in groups 2, 3 or 4, we add the index $j$ to $\mathcal{S}$ if $i \in \mathcal{S}$, otherwise we add $i$ to $\mathcal{S}$. Since we only consider internal observations when constructing $\mathcal{S}$, it is clear that $\mathcal{S}$ contains at most $d_1$ indices. We now construct the simulator. For clarity, observe that the MultLinear algorithm can be equivalently represented using the following matrix:

$$\begin{pmatrix}
a_0 \otimes g(a_0) & 0 & r_{0,1} & r_{0,2} & \cdots & r_{0,t} \\
a_1 \otimes g(a_1) & f(a_0,a_1) \ominus r_{0,1} & 0 & r_{1,2} & \cdots & r_{1,t} \\
a_2 \otimes g(a_2) & f(a_0,a_2) \ominus r_{0,2} & f(a_1,a_2) \ominus r_{1,2} & 0 & \cdots & r_{2,t} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
a_d \otimes g(a_d) & f(a_0,a_t) \ominus r_{0,t} & f(a_1,a_t) \ominus r_{1,t} & f(a_2,a_t) \ominus r_{2,t} & \cdots & 0
\end{pmatrix}$$

with $f(x,y) = x \otimes g(y) \oplus y \otimes g(x)$. In this setting, $c_{i,j}$ corresponds to the partial sum of the $j+2$ first elements of line $i$. For each variable $r_{i,j}$ or $r'_{i,j}$ entering in the

computation of an observation, we assign it a fresh random value. Then, for each observation in the first group, $a_i$ is provided to the simulator (by definition of $\mathcal{S}$) thus the observation is perfectly simulated. For each observation in the second group, $a_i$ is provided to the simulator and $r'_{i,j}$ is already assigned to a random value thus the observation is perfectly simulated. For each observation in the third group, we consider two cases. If $j \in \mathcal{S}$, then $a_j$ is also provided to the simulator and the observation can be perfectly simulated with $a_i$, $a_j$ and $r'_{i,j}$. If $j \notin \mathcal{S}$, then $r'_{i,j}$ does not enter in the computation of any other observation and $a_j \ominus r'_{i,j}$ can be assigned to a fresh random value. The observation is thus perfectly simulated with $a_i$. For each observation in the fourth group, we also consider two cases. If $j \in \mathcal{S}$, then $a_j$ is also provided to the simulator and the observation is perfectly simulated using $a_i$, $a_j$, $r_{i,j}$ and $r'_{i,j}$. If $j \notin \mathcal{S}$, then $r_{i,j}$ does not enter in the computation of any other observation. Thus, since this observation is invertible with respect to $r_{i,j}$ it can be perfectly simulated by a fresh random value. Finally, for each observation in the fifth group, we consider the different terms. The first product $a_i \otimes g(a_i)$ can be perfectly simulated with $a_i$. Then, the sum of $r_{i,j}$ can be perfectly simulated with the corresponding random values. As for the sum of $(a_i \otimes g(a_j) \oplus a_j \otimes g(a_i) \ominus r_{j,i})$ we consider two cases. If $j \in \mathcal{S}$, this sum can be perfectly simulated with $a_i$, $a_j$ and $r_{j,i}$. Otherwise, $r_{j,i}$ does not enter in the computation of any other observation and we can simulate the entire term using a fresh random value.

We still have to simulate the observed output values for rows on which no internal values are observed. Remark that simulating the $i$th line also necessarily fixed the value of all random variables appearing in the $i$th column (so that dependencies between variables are preserved). After internal observations are simulated, at most $d_1$ lines of the matrix are fully filled. Therefore, at least $t - d_1 \geqslant d_2$ (with $d_2 > 0$ if the adversary makes an output observation) random values are not yet simulated on lines on which no internal observations are made. For each output observation made on one such line (say $i$), we can therefore pick a different $r_{i,j}$ that we fix so that output $i$ can be simulated using a freshly sampled uniform value. □

*Remark 3.* Note that the first part of the proof, involving the simulation of internal observations only, was initially made in [8] to prove the gadget $t$-NI. However, the authors omitted one internal variable: $a_i \otimes g(r'_{i,j}) \ominus r_{i,j}$. We thus fix the proof of $t$-NI and further extend it to $t$-SNI by simulating outputs without any additional input shares.

*Remark 4.* In the compiler, we use the second algorithm provided by Coron et al. (Algorithm 5 in [8]) to compute the multiplication $x \otimes g(x)$ using a table. The algorithm is not exactly the same but the security proof is *a priori* similar. Namely, even if the intermediate variables are quite different, they can be classified like the ones of Algorithm 4 with the same dependencies (group 5 is exactly the same for both algorithms). Since we use these same dependencies in the aforementioned proof to explain the simulation of all possible sets of observations independently from the secret, we can reasonably claim that Algorithm 5 from [8] is also $t$-SNI.

## C Proof of Theorem 4.

The proof makes use of the following facts, that can be used to construct a simulator for $\mathsf{Robust}(F, G)$ and bound the size of the sets of shares of the inputs it requires. We easily extend the notions of gadget and (S/T)NI to include algorithms such as Half and Double that do not use the same $m$ on inputs and outputs.

– For any observation set $\Omega^{\mathsf{H}}$ on Half, there exists a $(2|\Omega^{\mathsf{H}}|)$-compatible input projection $\mathcal{S}^{\mathsf{H}}$ such that Half is $(\mathcal{S}^{\mathsf{H}}, \Omega^{\mathsf{H}})$-NI. Indeed, whenever share $i$ of the output is observed, we can give the simulator both shares $2i$ and $2i+1$ of the input.

– For an observation set $\Omega^{\mathsf{D}} = (\mathcal{I}, \mathcal{O})$ on Double, there exists a $(|\mathcal{I}| + |\mathcal{O}|)$-compatible input projection $\mathcal{S}^{\mathsf{D}}$ such that Double is $(\mathcal{S}^{\mathsf{D}}, \Omega^{\mathsf{D}})$-NI. Indeed, whenever both output shares $2i$ and $2i+1$ are observed, the simulator is given input share $i$, otherwise, no input share is needed; internal observations are exactly the inputs and are trivially simulated. □

*Proof (Theorem 4).* Let $\Omega^{G'} = ((\mathcal{I}_G^{\mathsf{R}}, \mathcal{I}^{\mathsf{H}}, \mathcal{I}^G), \mathcal{O}^{G'})$ be a $t$-admissible observation set on $G'$, and $\Omega^{F'} = ((\mathcal{I}^F, \mathcal{I}^{\mathsf{D}}, \mathcal{I}_F^{\mathsf{R}}), \mathcal{O}^{F'})$ be a $t$-admissible observation set on $F'$.

$\Omega^G = (\mathcal{I}^G, \mathcal{O}^{G'})$ is $t$-admissible, therefore there exists a $(|\mathcal{I}^G|+|\mathcal{O}^{G'}|)$-compatible $\mathcal{S}^G$ such that $G$ is $(\mathcal{S}^G, \Omega^G)$-NI.

By the property on Half, there exists a $(2(|\mathcal{I}^{\mathsf{H}}| + |\mathcal{S}^G|))$-compatible $\mathcal{S}^{\mathsf{H}}$ such that Half is $(\mathcal{S}^{\mathsf{H}}, (\mathcal{I}^{\mathsf{H}}, \mathcal{S}^G))$-NI.

Now, the observation set $\Omega_G^{\mathsf{R}} = (\mathcal{I}^{\mathsf{R}}, \mathcal{S}^{\mathsf{H}})$ on the Refresh is $2t$-admissible. Therefore, there exists a $|\mathcal{I}_G^{\mathsf{R}}|$-compatible $\mathcal{S}^{G'}$ such that this instance of Refresh is $(\mathcal{S}^{G'}, \Omega_G^{\mathsf{R}})$-NI. It is important to note here that $|\mathcal{S}^{G'}| \leq |\mathcal{I}_G^{\mathsf{R}}| \leq t$.

We can now move on to simulating $F'$, augmenting the observation set $\Omega^{F'}$ with the observations $\mathcal{S}^{G'}$ on its output necessary to perfectly simulate $G'$.

The observation set $\Omega_F^{\mathsf{R}} = (\mathcal{I}_F^{\mathsf{R}}, \mathcal{O}_F^{\mathsf{R}} \cup \mathcal{S}^{G'})$ is $2t$-admissible. Therefore, there exists a $|\mathcal{I}_F^{\mathsf{R}}|$-compatible input projection $\mathcal{S}_F^{\mathsf{R}}$ such that this instance of the Refresh gadget is $(\mathcal{S}_F^{\mathsf{R}}, \Omega_F^{\mathsf{R}})$-NI. Note again that $|\mathcal{S}_F^{\mathsf{R}}| \leq t$, here.

Considering the observation set $\Omega^{\mathsf{D}} = (\mathcal{I}^{\mathsf{D}}, \mathcal{S}^{\mathsf{R}})$ on the Double procedure, we deduce the existence of a $(|\mathcal{I}^{\mathsf{D}}|+|\mathcal{S}_F^{\mathsf{R}}|)$-compatible input projection $\mathcal{S}^{\mathsf{D}}$ (in fact, the bound is tighter than this, but we only need this one).

Finally, we use the fact that $F$ is $t$-TNI to finish constructing the simulator and bounding the size of the final input projection $\mathcal{S}$. □

### C.1 Application: Compilation of Stateful Circuits

We now consider the compilation of stateful circuits, or programs meant to be run in contexts where the adversary fully controls the rate at which public inputs are provided, and may therefore accommodate large break periods between executions (or loop iterations) during which he can move probes adaptively. Such stateful circuits can be compiled using the stateless compiler in combination with the robust mask refreshing operations described in Section 6, in a way similar to that of Ishai, Sahai and Wagner [15].

More explicitly, the secret state is initially shared uniformly over $m = t + 1$ shares (away from the adversary's observations), and is stored over $2m$ shares using

the $\mathsf{Double}; \mathsf{Refresh}_{2t+1}$ program (possibly in view of the adversary if desired, letting him place $t$ probes). Each oracle $G$ is compiled using our compiler from Section 7 (or any other techniques providing the desired level of security) into a gadget $\bar{G}$. Queries to oracle $G$ are then answered using the transformed oracle $\bar{G}' = \mathsf{Refresh}_{\mathsf{st}}; \mathsf{Half}_{\mathsf{st}}; \bar{G}; \mathsf{Double}_{\mathsf{st}}; \mathsf{Refresh}_{\mathsf{st}}$, where the $\mathsf{st}$ subscript indicates that the annotated gadget should be executed on each variable in the shared state. We let the adversary adaptively place $t$ probes during each run of $\bar{G}'$.

Following Theorem 4 (and its Corollary 1), it is clear that each adversary query's leakage can be perfectly simulated without having access to the initial secret, simply by composing the simulators and applying them to a uniformly sampled initial secret.

# D Details of the Type System

In Section 5, we present a simple typing judgment that is suitable to capture basic non-interference notions on core gadgets and their compositions. A simple type system could be devised to essentially mimic and automate the proof technique presented in Section 4. However, such a type system would fail to scale to large programs where a few non-affine components are reused many times and interleaved with complex linear operations. In such a setting, it is important for our verification tool to support incremental verification, where sub-procedures can be verified once, and their inferred type used many times in the verification of the full algorithm. On the other hand, Section 5 also illustrates through a simple example that obtaining incrementality without losing soundness or precision is challenging. We therefore consider here more complex typing judgments and formally relate them to those discussed in the body of the paper.

Our type system relies on two kinds of judgements: one expressing types on gadgets (used to type core gadgets, but also defined gadgets), and the other expressing types on statements and code.

We call *constraints* pairs of the form $\mathcal{C} = (\mathcal{C}^{\mathcal{O}}, \mathcal{C}^{\phi})$ with $\mathcal{C}^{\mathcal{O}} \subseteq \mathbb{O}$ and $\mathcal{C}^{\phi}$ a formula of the form $\bigwedge \phi(\mathcal{C}^{\mathcal{O}})$, where $\phi$ is a predicate expressing that $|\bigcup_{X \in \mathcal{C}^{\mathcal{O}}} X|$ can be bounded by a sum of cardinals of distinct symbols in $\mathbb{I}$. We call *inferred guarantees* (or simply *guarantees*) tuples of the form $\mathcal{G} = (\mathcal{G}^{\mathcal{I}}, \mathcal{G}^{\mathcal{S}}, \mathcal{G}^{\leq})$ with $\mathcal{G}^{\mathcal{I}} \subseteq \mathbb{I}$, $\mathcal{G}^{\mathcal{S}} \subseteq \mathbb{S}$, and $\mathcal{G}^{\leq}$ is a formula of the form $\bigwedge |\mathcal{S}| \leq |\mathcal{I}|$ where $\mathcal{I} \in \mathbb{I}$ and $\mathcal{S} \in \mathbb{S}$. Given an $(n, \ell, o)$ gadget $G$ and a constraint $\mathcal{C}$ and guarantee $\mathcal{G}$, we call *input types* $n$-tuples $\Delta_{\mathsf{in}}$ of set expressions in $\mathcal{C}^{\mathcal{O}} \cup \mathcal{G}^{\mathcal{I}} \cup \mathcal{G}^{\mathcal{S}}$, and *output types* $o$-tuples $\Delta_{\mathsf{out}}$ of set expressions in $\mathcal{C}^{\mathcal{O}}$ (in fact it is exactly $\mathcal{C}^{\mathcal{O}}$ seen as a tuple).

Essentially, when checking a side-condition in order to simulate a gadget, our type system collects those constraints about the current sub-procedures output observations that it has not successfully discharged into the constraint. Accumulated constraints are then used as a side-condition to the simulation of that sub-procedure, and therefore checked when type-checking its call.

**Judgments on Gadgets** Given a gadget $G$, constraint $\mathcal{C} = (\mathcal{C}^{\mathcal{O}}, \mathcal{C}^{\phi})$, guarantee $\mathcal{G} = (\mathcal{G}^{\mathcal{I}}, \mathcal{G}^{\mathcal{S}}, \mathcal{G}^{\leq})$ and input and output types $\Delta_{\mathsf{in}}$ and $\Delta_{\mathsf{out}}$, we write

$$\mathcal{C} \vdash G : \mathcal{G}, \Delta_{\mathsf{in}} \rightarrow \Delta_{\mathsf{out}}$$

to mean that, for all $\mathcal{O} \in [\![\mathcal{C}^{\mathcal{O}}]\!]$ such that $\mathcal{C}^{\phi}$ holds, and for all $\mathcal{I} \in [\![\mathcal{G}^{\mathcal{I}}]\!]$, there exists $\mathcal{S} \in [\![\mathcal{G}^{\mathcal{S}}]\!]$ such that $\mathcal{G}^{\leq}$ holds and $G$ is $(\Delta_{\mathsf{in}}, (\mathcal{G}^{\mathcal{I}}, \Delta_{\mathsf{out}}))$-$\mathsf{NI}$.

**Judgments on Code** Given a statement $p$, a constraint $\mathcal{C} = (\mathfrak{O}, \mathcal{C}_2^\phi)$, guarantees $\mathcal{G} = (\mathcal{G}^\mathcal{I}, \mathcal{G}^\mathcal{S}, \mathcal{G}^\leq)$ and $\mathcal{G}_2 = (\mathcal{G}_2^\mathcal{I}, \mathcal{G}_2^\mathcal{S}, \mathcal{G}_2^\leq)$, and typemaps $\Gamma_1$ and $\Gamma_2$, we write

$$\mathcal{C}_2^\phi, \mathcal{G}_2 \vdash^\mathfrak{O} p : \mathcal{G}, \Gamma_1 \to \Gamma_2$$

to mean (considering $p$ as a gadget from state to state): for all $\mathcal{O} \in [\![\mathfrak{O}]\!]$, for all $\mathcal{I}_2 \in [\![\mathcal{G}_2^\mathcal{I}]\!]$ and for all $\mathcal{S}_2 \in [\![\mathcal{G}_2^\mathcal{S}]\!]$ such that $\mathcal{C}_2^\phi$ and $\mathcal{G}_2^\leq$ hold, and for all $\mathcal{I} \in [\![\mathcal{G}^\mathcal{I}]\!]$, there exists $\mathcal{S} \in [\![\mathcal{G}^\mathcal{S}]\!]$ such that $\mathcal{G}^\leq$ holds and $p$ is $(\Gamma_1, (\mathcal{G}^\mathcal{I}, \Gamma_2))$-NI.

Axioms in our type system are typing judgments on affine gadgets and on the Refresh and SecMult gadgets. They are as follows:

- $(\{\mathcal{O}\}, \phi(|\mathcal{O}|)) \vdash \mathsf{Refresh} : (\{\mathcal{I}\}, \{\mathcal{S}\}, |\mathcal{S}| \leq |\mathcal{I}|), \mathcal{S} \to \mathcal{O}$
- $(\{\mathcal{O}\}, \phi(|\mathcal{O}|)) \vdash \mathsf{SecMult} : (\{\mathcal{I}\}, \{\mathcal{S}_1, \mathcal{S}_2\}, \bigwedge_{i \in \{1,2\}} |\mathcal{S}_i| \leq |\mathcal{I}|), (\mathcal{S}_1, \mathcal{S}_2) \to \mathcal{O}$

Propositions 1 and 2 justify these types. Indeed, they imply the existence of an input projection $\mathcal{S}$ that fulfills the desired non-interference property as soon as $|\mathcal{I} \cup \mathcal{O}| \leq t$. This follows from the condition $\phi(|\mathcal{O}|)$ and the global constraint that the total number of observations is bounded by $t$ (note that $\mathcal{I}$ is disjoint from all other observation sets). The guarantees are exactly the compatibility constraints on $\mathcal{S}$ stated in the Propositions. Similarly, an affine $(n, \ell, 1)$-gadget $G$ can be equipped with the following type, which is justified by theorem 1

$$(\{\mathcal{O}\}, \emptyset) \vdash G : (\{\mathcal{I}\}, \{\mathcal{S}\}, |\mathcal{S}| \leq |\mathcal{I}|), (\mathcal{S} \cup \mathcal{O})^n \to \mathcal{O}$$

**Typing rules.** Figure 9 presents the rules of our type system.

Rule (SEQ) is as expected. Typing the sequential composition $c_1; c_2$ proceeds as follows. First the intermediate typemap $\Gamma$ on variables occurring in $c_2$ and the corresponding sets of constraints $\mathcal{C}_2^\phi$ and guarantees $\mathcal{G}_2$ are computed from the desired output typemap $\Gamma_2$ and guarantees $\mathcal{G}$. Then, the input typemap $\Gamma_1$ on variables occurring in $c_1$ and the corresponding sets of constraints $\mathcal{C}_1^\phi$ and guarantees $\mathcal{G}_1$ are computed from the intermediate typemap $\Gamma$ and combined guarantees $\mathcal{G} \cup \mathcal{G}_2$. The final constraints and guarantees on $c_1; c_2$ are obtained by gathering those obtained on both sub-statements.

Rule (CALL) is the heart of the type system. When defining sub-gadgets, we identify their arguments and results by variable name, and consider that the body reads inputs directly from the indicated variables and writes output directly into result variables. To check a call we first check that the called gadget is well-typed (either by axiom if it is a core gadget, or by using Rule (GADGET) if it is user-defined). The set names in $\mathcal{G}^\mathcal{I}$ and $\mathcal{G}^\mathcal{S}$ are refreshed (to ensure unicity), and we check, using $\mathcal{G}_2 \vDash \mathcal{C}^\phi\{\mathcal{C}^\mathcal{O} \leftarrow \Gamma(\boldsymbol{x})\} \rightsquigarrow \mathcal{C}_2^\phi$, that the gadget's $\mathcal{C}^\phi$ constraints hold under $\mathcal{G}$ when the output names $\mathcal{C}^\mathcal{O}$ are instantiated with the types of $\boldsymbol{x}$, potentially producing a new set of constraints $\mathcal{C}_2^\phi$ on the caller's output type. When computing the call's input typemap $\Gamma'$, we reset the type of $\boldsymbol{x}$ (since its shares no longer need to be simulated) and associate to each output variable $y$ in $\boldsymbol{y}$ the union of its output type in $\Gamma$ and of its output type in gadget $G$.

Rule (GADGET) typechecks the code of a gadget $G$ assuming distinct and fresh output types on each variable. This produces a first set of constraints $\mathcal{C}_1^\phi$ and of guarantees

$\mathcal{G}$. From these and the computed input types on the gadget's input variables, we then compute a sufficient set of constraints on the output types to ensure that the guarantees are sufficient to imply the cardinality condition on input types. The constraints are conjoined to form a constraint on the gadget itself.

We now discuss how conditions of the form $\mathcal{G} \vDash \mathcal{C}^\phi\{\mathcal{C}^\mathcal{O} \leftarrow \Gamma(\boldsymbol{x})\} \rightsquigarrow \mathcal{C}_2^\phi$ are checked. After substituting $\Gamma(\boldsymbol{x})$ for $\mathcal{C}^\mathcal{O}$ in $\mathcal{C}^\phi$, the constraint is a conjunction of $\phi(\mathcal{E}_i)$, where $\mathcal{E}_i \in \mathcal{E}(\mathbb{O}' \cup \mathcal{G}^\mathcal{I} \cup \mathcal{G}^\mathcal{S})$, where $\mathbb{O}'$ is a subset of $\mathbb{O}$ disjoint from $\mathcal{C}^\mathbb{O}$. Since we know that the $\mathcal{O}$s are disjoint from the $\mathcal{I}$s and $\mathcal{S}$s, the constraint can be rewritten as a conjunction $\mathcal{C}^{\mathbb{O}'} = \bigwedge \phi(\mathcal{E}_i^{\mathbb{O}'})$ with $\mathcal{E}_i^{\mathbb{O}'} \in \mathcal{E}(\mathbb{O}')$ on the one hand, and a conjunction of $\mathcal{C}^\mathcal{G} = \phi(\mathcal{E}_i^\mathcal{G})$ with $\mathcal{E}_i^\mathcal{G} \in \mathcal{E}(\mathcal{G}^\mathcal{I} \cup \mathcal{G}^\mathcal{S})$ on the other. $\mathcal{C}^{\mathbb{O}'}$ is output as the new constraint on the caller's output type, and the guarantees in $\mathcal{G}$ are used to check whether $\mathcal{C}^\mathcal{G}$ holds.

$$\frac{\mathcal{C}_2^\phi, \mathcal{G} \vdash^\mathfrak{D} c_2 : \mathcal{G}_2, \Gamma \to \Gamma_2 \qquad \mathcal{C}_1^\phi, \mathcal{G} \cup \mathcal{G}_2 \vdash^\mathfrak{D} c_1 : \mathcal{G}_1, \Gamma_1 \to \Gamma}{\mathcal{C}_1^\phi \wedge \mathcal{C}_2^\phi, \mathcal{G} \vdash^\mathfrak{D} c_1; c_2 : \mathcal{G}_1 \cup \mathcal{G}_2, \Gamma_1 \to \Gamma_2} \quad (\textsc{Seq})$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash G : \mathcal{G}, \Delta_1 \to \Delta_2 \qquad \text{set names in } \mathcal{G}^\mathcal{I}, \mathcal{G}^\mathcal{S} \text{ are refreshed} \\ \Gamma' = \Gamma[\boldsymbol{x} \mapsto \emptyset] \bigsqcup_{\boldsymbol{y}} \Delta_1\{\mathcal{C}^\mathcal{O} \leftarrow \Gamma(\boldsymbol{x})\} \qquad \mathcal{G}_2 \vDash \mathcal{C}^\phi\{\mathcal{C}^\mathcal{O} \leftarrow \Gamma(\boldsymbol{x})\} \rightsquigarrow \mathcal{C}_2^\phi\end{array}}{\mathcal{C}_2^\phi, \mathcal{G}_2 \vdash^\mathfrak{D} \boldsymbol{x} \leftarrow G(\boldsymbol{y}) : \mathcal{G}, \Gamma' \to \Gamma} \quad (\textsc{Call})$$

$$\frac{\begin{array}{c}G = (\boldsymbol{x}, c_G, \boldsymbol{r}) \qquad \mathcal{C}_1^\phi, \emptyset \vdash^{\Delta_2} c_G : \mathcal{G}, \Gamma \to \bot[\boldsymbol{r} \mapsto \Delta_2] \\ \mathcal{G} \vDash \phi(\Gamma(\boldsymbol{x})) \rightsquigarrow \mathcal{C}_2^\phi \qquad \text{names in } \Delta_2 \text{ are distinct}\end{array}}{(\Delta_2, \mathcal{C}_1^\phi \wedge \mathcal{C}_2^\phi) \vdash G : \mathcal{G}, \Gamma(\boldsymbol{x}) \to \Delta_2} \quad (\textsc{Gadget})$$

Fig. 9: Typing rule for $t$-NI

**Theorem 5.** *Given an $(n, \ell, o)$-gadget $G$, if $\mathcal{C} \vdash G : \mathcal{G}, \Delta_{\mathsf{in}} \to \Delta_{\mathsf{out}}$ for some $\mathcal{G}$, $\Delta_{\mathsf{in}}$ and $\Delta_{\mathsf{out}}$, and for some $\mathcal{C}$ such that $\mathcal{C}^\phi = \phi(\mathcal{C}^\mathcal{O})$, then the following judgments (as defined in Section 5) are valid.*

$$\vdash_{\mathcal{C}^\mathcal{O}, \emptyset \uplus \mathcal{G}^\mathcal{I}, \emptyset \uplus \mathcal{G}^\mathcal{S}} \{\mathcal{G}^\leq; \Delta_{\mathsf{in}}\}\ G\ \{|\mathcal{C}^\mathcal{O}| \leq t; \Delta_{\mathsf{out}}\} \tag{3}$$

$$\mathcal{C}^\mathcal{O}; \mathcal{G}^\mathcal{I}; \mathcal{G}^\mathcal{S}; \mathcal{G}^\leq \vDash_G \Delta_{\mathsf{in}} \tag{4}$$

*Therefore, $G$ is $t$-NI.*

## E Input/Generated code for AES S-box

The following codes show the input code sent to the compiler (AES S-Box) and the resulting generated code. Notice that the compiler automatically introduces the needed refresh (in function `pow254`) and use the affine gadget `and_pub` instead of the one based on SecMult.

AES S-Box input code

```
bint8_t pow254(bint8_t x){
  bint8_t r, w, z;




  z  = gf256_pow2(x);

  r  = gf256_mul(x,z);
  w  = gf256_pow4(r);

  r  = gf256_mul(r,w);
  r  = gf256_pow16(r);
  r  = gf256_mul(r,w);
  r  = gf256_mul(r,z);
  return r;
}

bint8_t affine (bint8_t a){
  uint8_t j;
  bint8_t res = 0x00;
  bint8_t t;



  for (j = 0; j < 8; j++){
    t    = a >> j;
    t   ^= a >> ((j + 4)%8);
    t   ^= a >> ((j + 5)%8);
    t   ^= a >> ((j + 6)%8);
    t   ^= a >> ((j + 7)%8);
    t   &= 0x01;
    res ^= t << j;
  }




  res = res ^ 0x63;
  return res;
}

bint8_t sbox(bint8_t a){
  bint8_t r;

  r = pow254(a);
  r = affine(r);
  return r;
}
```

Generated code

```
void pow254 (bint8_t r, bint8_t x){
  bint8_t x0;
  bint8_t w;
  bint8_t z;
  bint8_t aux;
  bint8_copy(x0, x);
  gf256_pow2(z, x0);
  bint8_refresh(aux, z);
  gf256_mul(r, x0, aux);
  gf256_pow4(w, r);
  bint8_refresh(aux, w);
  gf256_mul(r, r, aux);
  gf256_pow16(r, r);
  gf256_mul(r, r, w);
  gf256_mul(r, r, z);
  return;
}

void affine (bint8_t res, bint8_t a){
  bint8_t aux;
  bint8_t a0;
  uint8_t j;
  bint8_t t;
  bint8_copy(a0, a);
  bint8_to_base(res, 0);
  for(j = 0; j < 8; j++) {
    bint8_rshift(t, a0, j);
    bint8_rshift(aux, a0, (j+4)%8);
    bint8_xor(t, t, aux);
    bint8_rshift(aux, a0, (j+5)%8);
    bint8_xor(t, t, aux);
    bint8_rshift(aux, a0, (j+6)%8);
    bint8_xor(t, t, aux);
    bint8_rshift(aux, a0, (j+7)%8);
    bint8_xor(t, t, aux);
    bint8_and_pub(t, t, 1);
    bint8_lshift(aux, t, j);
    bint8_xor(res, res, aux);
  }
  bint8_xor_pub(res, res, 99);
  return;
}

void sbox (bint8_t r, bint8_t a){
  bint8_t a0;
  bint8_copy(a0, a);
  pow254(r, a0);
  affine(r, r);
  return;
}
```