Strong Non-Interference and Type-Directed Higher-Order Masking*

Gilles Barthe
IMDEA Software Institute
Madrid, Spain

Sonia Belaïd
Thales Communications & Security
Gennevilliers, France

François Dupressoir
IMDEA Software Institute
Madrid, Spain

Pierre-Alain Fouque
Université de Rennes 1
Rennes, France

Benjamin Grégoire
Inria Sophia-Antipolis – Méditerranée
Sophia-Antipolis, France

Pierre-Yves Strub
IMDEA Software Institute
Madrid, Spain

Rébecca Zucchini
Inria Sophia-Antipolis – Méditerranée
École Normale Supérieure de Cachan
France

ABSTRACT

Differential power analysis (DPA) is a side-channel attack in which an adversary retrieves cryptographic material by measuring and analyzing the power consumption of the device on which the cryptographic algorithm under attack executes. An effective countermeasure against DPA is to *mask* secrets by probabilistically encoding them over a set of shares, and to run *masked* algorithms that compute on these encodings. Masked algorithms are often expected to provide, at least, a certain level of *probing security*.

Leveraging the deep connections between probabilistic information flow and probing security, we develop a precise, scalable, and fully automated methodology to verify the probing security of masked algorithms, and generate them from unprotected descriptions of the algorithm. Our methodology relies on several contributions of independent interest, including a stronger notion of probing security that supports compositional reasoning, and a type system for enforcing an expressive class of probing policies. Finally, we validate our methodology on examples that go significantly beyond the state-of-the-art.

1. INTRODUCTION

Differential power analysis, or DPA [26], is a class of side-channel attacks in which an adversary extracts secret data from the power consumption of the device on which a program manipulating the data executes. One practical countermeasure against DPA, called *masking* [12, 23], transforms an algorithm that performs computations over a finite ring $\mathbb{K}$ into a randomized algorithm that manipulates probabilistic encodings.[1] At an abstract level, any masking transformation performs two tasks. First, it replaces every algebraic operation performed by the original algorithm by a call to a *gadget*, i.e. a probabilistic algorithm that simulates the behavior of algebraic operations on probabilistic encodings. Second, it inserts *refreshing* gadgets, i.e. gadgets that take a probabilistic encoding of v and rerandomizes its shares in order to produce another probabilistic encoding w of v. Inserting refreshing gadgets does not change the functional behavior of the masked algorithm, and increases the randomness complexity and execution time of the masked program. However, it is also compulsory for achieving security. Therefore, an important line of research is to find suitable trade-offs that ensure security while minimizing the performance overhead of masking; see [9] for recent developments in this direction.

The baseline notion of security for masked algorithms is t-probing security. Informally, an algorithm P is t-probing secure if the values taken by at most t intermediate variables of P during execution do not leak any information about secrets (held by its inputs). More formally, an algorithm P achieves t-probing security iff for every set of at most t intermediate variables, the joint distributions of the values taken by these intermediate variables coincide for any two executions initiated from initial inputs that agree on t shares of each input encoding. Stated in this form, probing security is an instance of probabilistic information flow, universally quantified over all position sets that meet a cardinality constraint, and is therefore potentially amenable to formal analysis using a well-developed body of work on language-based security and program verification. Indeed, the connection between probing security and information flow has been instrumental in a promising line of research, initiated in [28] and further developed in [8, 21, 20, 4], which uses type systems, program logics, SMT solvers and other methods for verifying

*Preliminary and long versions of this work appear as revisions of IACR ePrint report 2015/506 [5].

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978427

[1] A t-encoding of an element $v \in \mathbb{K}$ is a $(t+1)$-tuple $\mathbf{v} = \langle \mathbf{v}^0, \ldots, \mathbf{v}^t \rangle$ such that $[\![\mathbf{v}]\!] \triangleq \mathbf{v}^0 \oplus \ldots \oplus \mathbf{v}^t = v$. Each of the $\mathbf{v}^i \in \mathbb{K}$ in an encoding $\mathbf{v}$ of v is called a *share*. Moreover, t is called the masking order. A probabilistic encoding of v is a distribution over encodings of v.

or synthesizing masked algorithms at small (≤ 5) orders. However, none of these works addresses the problem of composition, and all fail to scale either to higher orders or to larger algorithms.

Contributions. We develop precise and scalable techniques for synthesizing masked algorithms that achieve probing security. Our techniques apply to a wide range of probing policies, including existing policies and new policies defined in this paper, and deliver masked algorithms that outperform (in terms of randomness complexity and computational efficiency) prior approaches. In more detail, we make the following broad contributions:

1. *Strong non-interference.* We introduce a stronger notion of probing security, which we call strong non-interference, and prove that it is in fact satisfied by many (but not all) gadgets from the literature. Furthermore, we justify that strong non-interference is the desired property for refreshing gadgets, by reconsidering known negative and positive results [16] for a simplified example extracted from Rivain and Prouff's inversion algorithm [32]. We first observe that the refreshing gadget used in the original, flawed, algorithm does not enjoy strong non-interference. Second, we note that the refreshing gadget used in the fixed, secure, algorithm is indeed strongly non-interfering, and we show that one can prove the probing security of the fixed algorithm, based simply on the assumption that the refreshing gadget is strongly non-interfering. Generalizing these observations, we prove that every non-interfering algorithm can be turned into a strongly non-interfering algorithm, by processing its inputs or its output with a strongly non-interfering refreshing gadget. We also provide more general results about the composition of strongly non-interfering gadgets.

2. *Formal proofs.* We develop and implement an automated method, inspired from [4], for checking strong non-interference. We apply our automated verifier for strong non-interference to several gadgets from the literature and some other interesting compositions, for orders $t \leq 6$. For several more widely-used gadgets, we further use EasyCrypt [6] to provide machine-checked proofs of t-probing security for all t.

3. *Type-based enforcement of probing security.* We define an expressive language for specifying a large class of non-interference properties with cardinality constraints. Our language can be seen as a variant of the first-order theory of finite sets with cardinality constraints [33, 3], and can be used to specify baseline probing security and strong non-interference, among others. Then, we define a type system that enforces probing policies and prove its soundness. Furthermore, we show how to model in our language of probing policies the notion of affine gadget, and we show how it helps improve the precision of type-checking.

4. *Certifying Masking Transformation.* As a proof of concept, we implement a type inference algorithm and a certifying masking transformation that takes as input an arithmetic expression and returns a masked algorithm typable by our type system.[2] Our transformation improves over prior works by selectively inserting refreshing gadgets only at points where type-checking would otherwise fail. This strategy leads to improved efficiency while retaining provable soundness.

5. *Practical evaluation.* We evaluate our type system and masking transformation on complete algorithms at various orders, often achieving provable t-probing security levels far beyond the state-of-the-art for algorithms of those sizes, and with better performance than most known (provably secure) algorithms in terms of time, memory and randomness complexity.

Related work. Section 9 discusses related work in more detail. Here we focus on recent work on automated tools for the verification of synthesis of masked algorithms, starting with Moss et al. [28], who point out and leverage connections between probing security and probabilistic information-flow for first-order boolean masking schemes. Subsequent works in this direction accommodate higher-order and arithmetic masking, using type systems and SMT solvers [8], or model counting and SMT solvers [21, 20]. Although approaches based on model counting are more precise than early approaches based on type systems and can be extended to higher-order masking schemes, their algorithmic complexity constrains their applicability. In particular, existing 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 they consider (for instance, only analyzing a single round of Keccak at order 1, and algorithms for field operations at orders 2 and higher). Breaking away from model counting, Barthe et al. [4] develop efficient algorithms for analyzing the security of masked algorithms in the probing model. Their approach outperforms previous work and can analyze a full block of AES at first-order, reduced-round (4 rounds) AES at the second-order, and several S-box computation algorithms masked at the third and fourth orders. However, their work does not readily scale either to higher orders or to larger algorithms, mainly due to the lack of composition results.

Our work also bears some connections with language-based security, and in particular with work on the specification and the enforcement of confidentiality policies using techniques from programming languages. For instance, our work has similarities with the work of Pettai and Laud [30], who develop techniques for proving security of multi-party computations in the presence of strong adversaries, and work by Zdancewic et al. [34], who propose a compiler that partitions programs for secure distributed execution.

Mathematical preliminaries. A function $\mu : B \to \mathbb{R}^{\geq 0}$ is a (discrete) *distribution* over B if the subset $\mathsf{supp}(\mu)$ of B with non-zero weight under μ is discrete and moreover $\sum_{b \in \mathsf{supp}(\mu)} \mu(b) = 1$. We let $\mathcal{D}(B)$ denote the set of discrete distributions over B. Equality of distributions is defined as pointwise equality of functions. Distributions can be given a monadic structure with the two operators $\mathsf{munit}(\cdot)$ and $\mathsf{mlet}\ \cdot = \cdot$. For every $b \in B$, $\mathsf{munit}(b)$ is the unique distribution μ such that $\mu(b) = 1$. Moreover, given $\mu : \mathcal{D}(B)$ and $M : B \to \mathcal{D}(C)$, $\mathsf{mlet}\ x = \mu\ \mathsf{in} M\ x$ is the unique distribution μ' over C such that $\mu'(c) = \sum_b \mu(b)\ M(b)(c)$.

We often use the notion of marginals. The first and second marginals of a distribution $\mu \in \mathcal{D}(B_1 \times B_2)$ are the distributions $\pi_1(\mu) \in \mathcal{D}(B_1)$ and $\pi_2(\mu) \in \mathcal{D}(B_2)$ given by

$$\pi_1(\mu)(b_1) = \sum_{b_2 \in B_2} \mu(b_1, b_2) \qquad \pi_2(\mu)(b_2) = \sum_{b_1 \in B_1} \mu(b_1, b_2).$$

The notion of marginal readily extends to distributions over finite maps (rather than pairs).

2. A BIRD'S EYE VIEW OF STRONG NON-INTERFERENCE

Before formalizing our definitions, we give an intuitive description of our language for gadgets and of our security notions, based on simple examples.

[2]The cryptography literature often refers to such transformations as *masking compilers*. We purposely avoid this terminology, since the terms is used in programming languages for transformations that output executable code

Gadgets and Positions. Gadget $\mathsf{RefreshM}_2$ (Gadget 1) shows the description in our language of a mask refreshing gadget for $t = 2$. The gadget takes as input an encoding variable $\mathbf{a} \in \mathbb{K}^3$, where $\mathbb{K}$ is some finite ring and returns a new encoding $\mathbf{c} \in \mathbb{K}^3$ such that $[\![\mathbf{a}]\!] = [\![\mathbf{c}]\!]$. The gadget first makes local copies of individual input shares $\mathbf{a}^\imath$ (for $0 \leq \imath \leq 2$) of $\mathbf{a}$ into local variables $c_\imath$ (for $0 \leq \imath \leq 2$). After this first step, we sample uniform random elements from $\mathbb{K}$ into a local variable r and perform some ring operations. Finally, the algorithm returns a vector in $\mathbb{K}^3$, constructed from the final value of local variables c_0, c_1 and c_2.

Gadget 1 SNI Mask Refreshing with $t = 2$

function $\mathsf{RefreshM}_2(\mathbf{a})$
$c_{0,0} \leftarrow \mathbf{a}^0$; $c_{1,0} \leftarrow \mathbf{a}^1$; $c_{2,0} \leftarrow \mathbf{a}^2$;
$r_0 \xleftarrow{\$} \mathbb{K}$; $c_{0,1} \leftarrow c_{0,0} \oplus r_0$; $c_{1,1} \leftarrow c_{1,0} \ominus r_0$;
$r_1 \xleftarrow{\$} \mathbb{K}$; $c_{0,2} \leftarrow c_{0,1} \oplus r_1$; $c_{2,1} \leftarrow c_{2,0} \ominus r_1$;
$r_2 \xleftarrow{\$} \mathbb{K}$; $c_{1,2} \leftarrow c_{1,1} \oplus r_2$; $c_{2,2} \leftarrow c_{2,1} \ominus r_2$;
return $\langle c_{0,2}, c_{1,2}, c_{2,2} \rangle$

Note that the gadget is written in *single static assignment* (SSA) form, an intermediate representation in which each variable is defined exactly once. Having gadgets written in SSA form allows us to easily refer to the value of a particular variable at a particular point in the program–simply by referring to its name, which corresponds to a unique definition. In this paper, we refer to *positions* in gadgets and algorithms, which correspond exactly to intermediate variables. We distinguish between three different kinds of positions: *input positions*, which correspond to shares of the gadget's input (here, $\mathbb{I}_{\mathsf{RefreshM}_2} = \{\mathbf{a}^0, \mathbf{a}^1, \mathbf{a}^2\}$), *output positions*, which correspond to the variables that appear in the gadget's return vector (here, $\mathbb{O}^{\text{int}}_{\mathsf{RefreshM}_2} = \{c_{0,2}, c_{1,2}, c_{2,2}\}$), and *internal positions*, which refer to all other positions (here, $\mathbb{O}^{\text{ext}}_{\mathsf{RefreshM}_2} = \{c_{0,0}, c_{1,0}, c_{2,0}, c_{0,1}, c_{1,1}, c_{2,1}, r_0, r_1, r_2\}$). Intuitively, this separation allows us to distinguish between direct observations made by the adversary into a gadget (as *internal* positions), output shares about which the adversary may have learned some information by probing gadgets that use them as input (as *output* positions), and shares of the gadget's inputs (as *input* positions) about which the adversary is now learning information. In the following, we often write "the joint distribution of a set of positions" to discuss the joint distribution of the variables defined at these positions in the gadget (in order). For example, referring to $\mathsf{RefreshM}_2$, the joint distribution of the ordered set $\mathcal{O} = \langle c_{0,1}, c_{2,2} \rangle$ of positions can be described as the following function of $\mathbf{a}$, where we use $\$$ to denote a fresh uniform random sample in $\mathbb{K}$ (using indices to denote distinct samples): $[\![\mathsf{RefreshM}_2]\!]_{\mathcal{O}}(\mathbf{a}) \triangleq \langle \mathbf{a}^0 \oplus \$_0, (\mathbf{a}^2 \ominus \$_1) \ominus \$_2 \rangle$.

Probing Security and Non-Interference. The $\mathsf{RefreshM}_2$ gadget is known to be 2-probing secure, or 2-non-interfering (2-NI) in our terminology, in the sense that the joint distribution of any set of at most 2 of its positions, corresponding to adversary *probes*, depends on at most 2 shares of the gadget's inputs. This guarantees, if the input encoding is uniform, that no information about it leaks through any 2 probes in the circuit.

Considering again the set $\mathcal{O} = \langle c_{0,1}, c_{2,2} \rangle$ of positions and its distribution $[\![\mathsf{RefreshM}_2]\!]_{\mathcal{O}}$, it is easy to see–purely syntactically–that it depends, at most, on shares $\mathbf{a}^0$ and $\mathbf{a}^2$ of the gadget's input encoding. Similarly considering all possible pairs of positions, we can prove that each of them has a joint distribution that depends on at most two shares of $\mathbf{a}$.

Strong Non-Interference. Probing security is generally not *composable*: combining t-probing secure gadgets does not necessarily yield a t-probing secure algorithm [16]. Our main contribution is a new and stronger notion of security for gadgets, which we dub *strong non-interference* (or SNI), which does support some compositional reasoning. SNI reinforces probing security by requiring that the number of input shares on which the distribution of a given position set may depend be determined only by the number of *internal* positions present in that set. For example, consider again position set $\mathcal{O} = \langle c_{0,1}, c_{2,2} \rangle$ in $\mathsf{RefreshM}_2$, and note that it contains only one internal position ($c_{0,1}$). We have seen that the joint distribution $[\![\mathsf{RefreshM}_2]\!]_{\mathcal{O}}$ of that position set syntactically depends on two shares of $\mathbf{a}$. However, it can be equivalently expressed as $[\![\mathsf{RefreshM}_2]\!]_{\mathcal{O}}(\mathbf{a}) = \langle \$_0, (\mathbf{a}^2 \ominus \$_1) \ominus \$_2 \rangle$ (since the ring addition $\oplus$ is a bijection of each of its arguments and $\$_0$ is a fresh and uniform ring element). This shows that the distribution in fact depends on at most one share of $\mathbf{a}$ (here $\mathbf{a}^2$). In fact, it can be shown that $\mathsf{RefreshM}_2$ is 2-SNI. More generally, surprisingly many gadgets from the literature achieve SNI.

However, and not unexpectedly, some gadgets from the literature do not satisfy SNI. Consider for instance $\mathsf{RefreshA}_2$ (Gadget 2). It is easy to see that the gadget is 2-NI (each position $c_{\imath,\jmath}$ depends only on input share $\mathbf{a}^\imath$, and each position r_i is completely independent from the input encoding). Still, looking at position set $\mathcal{O}' = \langle c_{0,1}, c_{1,1} \rangle$, which is composed of one internal position and one external one, we see that the distribution $[\![\mathsf{RefreshA}_2]\!]_{\mathcal{O}'} \triangleq \langle \mathbf{a}^0 \oplus \$_0, \mathbf{a}^1 \oplus \$_0 \rangle$ does depend on more than one share of $\mathbf{a}$. $\mathsf{RefreshA}_2$ is therefore not 2-SNI.

Gadget 2 NI Mask Refreshing with $t = 2$

function $\mathsf{RefreshA}_2(\mathbf{a})$
$c_{0,0} \leftarrow \mathbf{a}^0$; $c_{1,0} \leftarrow \mathbf{a}^1$; $c_{2,0} \leftarrow \mathbf{a}^2$;
$r_0 \xleftarrow{\$} \mathbb{K}$; $c_{0,1} \leftarrow c_{0,0} \oplus r_0$; $c_{1,1} \leftarrow c_{1,0} \ominus r_0$;
$r_1 \xleftarrow{\$} \mathbb{K}$; $c_{0,2} \leftarrow c_{0,1} \oplus r_1$; $c_{2,1} \leftarrow c_{2,0} \ominus r_1$;
return $\langle c_{0,2}, c_{1,1}, c_{2,1} \rangle$

Compositional Probing Security. This small difference between NI and SNI has a significant effect on security when used in larger circuits. Indeed, the output positions of a strongly non-interfering gadgets do not depend on any of its input positions: when considered independently from internal positions (in the absence of internal probes), their distribution is uniform; and in the presence of internal probes, their joint distribution is entirely determined by that of the probed internal positions. This is essential in supporting *compositional* reasoning about the probing security of larger algorithms. In particular, this makes algorithms of the form shown in Algorithm 3 (for some gadgets R and G of the appropriate types that work on 2-encodings) easy to prove t-NI if R is $\mathsf{RefreshM}_2$, and illustrates why composition might fail if R is instantiated with $\mathsf{RefreshA}_2$.

Alg. 3 An abstract algorithm

function $\mathsf{Alg}_2(\mathbf{a})$
$\mathbf{b} := \mathsf{R}(\mathbf{a})$;
$\mathbf{c} := \mathsf{G}(\mathbf{a}, \mathbf{b})$;
return $\mathbf{c}$

A key observation to make is that an adversary that observes 2 positions internal to G may learn 2 shares of both $\mathbf{a}$ and $\mathbf{b}$. If R is instantiated with $\mathsf{RefreshA}_2$ (and is thus only 2-probing secure), the

2 shares of $\mathbf{b}$ can be used to infer information about 2 further shares of $\mathbf{a}$, which may give the adversary full knowledge of all 3 shares of $\mathbf{a}$. On the other hand, if R is instantiated with $\mathsf{RefreshM}_2$ (and is thus 2-SNI), the adversary's knowledge of 2 shares of $\mathbf{b}$ does not propagate any further back to $\mathbf{a}$, and the algorithm remains secure.

Broader uses of SNI. The notion of strong non-interference, and the masking transformation we define here have already found applications in follow-up work. Belaïd et al. [9] prove using our compositional techniques that their new non-interfering multiplication can be securely combined with the strongly non-interfering one of Rivain and Prouff [32] to build a strongly non-interfering AES S-box with reduced randomness complexity. Similarly, Goudarzi and Rivain [24] use our method to ensure the compositional security of their bitsliced software implementation of AES. Battistelo et al. [7] use and prove t-SNI for their $O(n \cdot \log n)$ mask refreshing gadget, allowing further randomness complexity reductions without loss of probing security. Coron et al. [17] use and prove t-SNI for their efficient parallel algorithms for the evaluation of SBoxes.

Outline. The rest of the paper is organized as follows. Section 3 formalizes our two-tier language for masked gadgets and algorithms, the notion of position, and their semantics, as well as the joint distribution of a set of positions. Sections 4, and 5 formalize probing security as t-non-interference, and formally define our new notion of t-strong-non-interference before illustrating it more generally with simple examples. In Section 6, we define a language to describe probing policies, and define a simple type system for enforcing probing policies of algorithms, formalizing and generalizing the simple compositional arguments outlined here. In Section 7, we present an automated method to verify the strong non-interference of arbitrary gadgets at small fixed orders, that follows the approach used above in arguing that $\mathsf{RefreshM}_2$ is 2-SNI, and adapts algorithms by Barthe et al. [4] to reduce the number of position sets to consider. In Section 8, we extend our type system into a masking transformation which automatically builds a masked algorithm from an unprotected program, carefully choosing the proper locations for strongly non-interfering refreshing gadgets. We evaluate on full cryptographic algorithms the performance of the type system, of the resulting transformation, and of the transformed algorithms. Section 9 discusses related work on leakage models, composition for probing security, and other masking transformations. We interleave discussions of interesting leads for future work.

3. MASKED ALGORITHMS

The formal development of this paper is based on a minimalist 2-tier language.[3] The lower tier models gagdets as sequences of probabilistic and (three-address code) deterministic assignments, whereas the upper tier models algorithms as sequences of gadget calls (we assume that each gadget call is tagged with its instruction number $\ell \in \mathbb{N}$). The formal definition of the language is given in Figure 1, where we use vector notations $(\vec{x}, \dots)$ to denote $(t+1)$-tuples of scalar variables, $\imath$ to denote indices (such that $0 \leq \imath \leq t$) in such a tuple or in encoding variables, and exponents $\cdot^\imath$ to denote the projection of a component out of a $(t+1)$-tuple (for example $\mathbf{a}^\imath$, or $\vec{x}^\imath$). We require gadgets and algorithms to be well-formed,

[3]However, the verification tool supports richer settings to which the theory extends smoothly and our examples are written in a more general language, closer to our implementation, that supports static **for** loops, direct assignments to shares ($\mathbf{a}^\imath \leftarrow e$), arbitrary expressions on the right-hand side of assignments, and a broader return syntax. For example, Gadget 4 shows generic descriptions of the mask refreshing algorithms from Section 2.

algorithm	$P(\mathbf{a}_1, \dots, \mathbf{a}_n)$	$::= s;$ return $\mathbf{a}$	
alg. body	s	$::= \mathbf{b} :=_\ell G(\mathbf{a}_1, \dots, \mathbf{a}_n)$	gadget call.
		$\mid s; s$	call seq.
gadget	$G(\mathbf{a}_1, \dots, \mathbf{a}_n)$	$::= c;$ return $\vec{x}$	
gadget body	c	$::= x \xleftarrow{\$} \mathbb{K}$	prob. assign.
		$\mid x \leftarrow e$	det. assign.
		$\mid c; c$	assign. seq.
expressions	e	$::= x, y, \dots$	variable
		$\mid \mathbf{a}^\imath$	$\imath^{\text{th}}$-share of $\mathbf{a}$
		$\mid x \star y$	ring operation

Figure 1: Syntax of masked algorithms

in the following sense. A gadget G is *well-formed* if its body is in SSA form, i.e. its scalar variables appear at most once on the left-hand side of an assignment. An algorithm P is well-formed if all its gadgets are defined and well-formed, and if, in all gadget calls $\mathbf{b} := G(\mathbf{a}_1, \dots, \mathbf{a}_n)$, variables $\mathbf{b}, \mathbf{a}_1, \dots, \mathbf{a}_k$ are pairwise disjoint.

We now turn to the semantics of gadgets and algorithms. Crucially, the semantics of gadgets and algorithms is instrumented to keep track of the joint distribution of *all* intermediate values computed during execution. Formally, we assume that scalar and encoding variables take values in $\mathbb{K}$ and $\mathbb{K}^{t+1}$, where $\mathbb{K}$ is the carrier set of a finite ring $(\mathbb{K}, 0, 1, \oplus, \ominus, \odot)$. We let $\mathsf{Val} = \mathbb{K}^{t+1}$ denote the set of encoded values. Furthermore, we let $\mathcal{A}$ denote the set of encoding variables and define the set of global memories as $\mathsf{Mem} = \mathcal{A} \to \mathbb{K}^{t+1}$. Likewise, we let $\mathcal{V}$ denote the set of scalar variables and define the set of local memories as $\mathsf{LMem} = \mathcal{V} \rightharpoonup \mathbb{K}$ and extended local memories as $\mathsf{ELMem} = (\mathbb{N} \times \mathcal{V}) \rightharpoonup \mathbb{K}$. Then, the semantics of a gagdet G is a function $[\![G]\!]$ that takes as input a global memory and returns a distribution over pairs of local memories and values. Likewise, the semantics of an algorithm P is a function $[\![P]\!]$ that takes as input a global memory and returns a distribution over extended local memories and values. The semantics is outlined in Figure 2.

In order to define probing security, we first define a notion of position that corresponds to the intuition illustrated in Section 2. First, we define the set $\mathbb{I} \triangleq \{\mathbf{a}^\imath \mid \mathbf{a} \in \mathcal{A}, 0 \leq \imath \leq t\}$ of input positions (these correspond to shares of encodings used in the gadget or algorithm), the set $\mathbb{O} \triangleq \mathbb{I} \cup \mathcal{V}$ of positions (composed of input positions and scalara variables) and the set $\mathbb{O}^+ \triangleq \mathbb{I} \cup (\mathbb{N} \times \mathcal{V})$ of extended positions (where scalar variables are tagged with a label in $\mathbb{N}$ to differentiate between uses of a variable in different gadgets). The input positions of a gadget G and of an algorithm P are denoted by $\mathbb{I}_G$ and $\mathbb{I}_P$ respectively and contain exactly those elements of $\mathbb{I}$ that correspond to encoding variables that occur in G or P. Likewise, the set of positions of a gadget G and of an algorithm P are denoted by $\mathbb{O}_G \subseteq \mathbb{O}$ and $\mathbb{O}_P \subseteq \mathbb{O}^+$ respectively and consist of all positions that occur in a gadget G, and all extended positions that occur in an algorithm P.

To capture the joint distribution of a set of positions $\mathcal{O}$ in a gadget G or an algorithm P (with $\mathcal{O} \subseteq \mathbb{O}_G$, resp. $\mathcal{O} \subseteq \mathbb{O}_P$), we take the marginal of the gadget or algorithm's semantics with respect to $\mathcal{O}$. These are denoted by $[\![G]\!]_\mathcal{O} : \mathsf{Mem} \to \mathcal{D}(\mathcal{O} \to \mathbb{K})$ and $[\![P]\!]_\mathcal{O} : \mathsf{Mem} \to \mathcal{D}(\mathcal{O} \to \mathbb{K})$ respectively.[4]

[4]In order to justify that the marginals have the required type, observe that one can refine the type of $[\![G]\!]$ given in Figure 2 to $\mathsf{Mem} \to$

$$
\begin{array}{rcl}
[\![e]\!](m, lm) & : & \mathbb{K} \qquad \text{with } m \in \mathsf{Mem} \text{ and } lm \in \mathsf{LMem} \\
[\![x]\!](m, lm) & = & lm(x) \\
[\![\mathbf{a}^i]\!](m, lm) & = & m(\mathbf{a})^i \\
[\![x \star y]\!](m, lm) & = & lm(x) \star lm(y) \\
\hline
[\![c]\!](m, lm) & : & \mathcal{D}(\mathsf{Mem} \times \mathsf{LMem}) \qquad \text{with } m \in \mathsf{Mem} \text{ and } lm \in \mathsf{LMem} \\
[\![x \leftarrow e]\!](m, lm) & = & \mathsf{munit}(m, lm\{x \leftarrow [\![e]\!](m, lm)\}) \\
[\![x \xleftarrow{\$} \mathbb{K}]\!](m, lm) & = & \mathsf{mlet}\ v = \mathcal{U}_{\mathbb{K}}\ \mathsf{in}\ \mathsf{munit}(m, lm\{x \leftarrow v\}) \\
[\![c_1; c_2]\!](m, lm) & = & \mathsf{mlet}\ (m_1, lm_1) = [\![c_1]\!](m, lm)\ \mathsf{in}\ [\![c_2]\!](m_1, lm_1) \\
\hline
[\![G]\!](m) & : & \mathcal{D}(\mathsf{LMem} \times \mathsf{Val}) \qquad \text{with } m \in \mathsf{Mem} \text{ and } G(\mathbf{a}_1, \ldots, \mathbf{a}_n) ::= c; \mathsf{return}\ \vec{x} \\
[\![G]\!](m) & = & \mathsf{mlet}\ (m_1, lm_1) = [\![c]\!](m, \emptyset)\ \mathsf{in}\ \mathsf{munit}(lm_1, lm_1(\vec{x})) \\
\\
\hline
[\![s]\!](m, elm) & : & \mathcal{D}(\mathsf{Mem} \times \mathsf{ELMem}) \qquad \text{with } m \in \mathsf{Mem},\ elm \in \mathsf{ELMem} \text{ and } G(\mathbf{a}_1, \ldots, \mathbf{a}_n) ::= c; \mathsf{return}\ \vec{x} \\
[\![\mathbf{b} :=_\ell G(\mathbf{c}_1, \ldots, \mathbf{c}_n)]\!](m, elm) & = & \mathsf{mlet}\ (lm_1, v) = [\![G]\!](m\{\mathbf{a}_0, \ldots, \mathbf{a}_t \leftarrow m(\mathbf{c}_0), \ldots, m(\mathbf{c}_t)\})\ \mathsf{in}\ \mathsf{munit}(m\{\mathbf{b} \leftarrow v\}, elm \uplus elm_1) \\
& & \qquad \text{where } elm_1 \text{ is the map defined by setting only } elm_1(\ell, v) = lm(v) \text{ for all } v \in \mathsf{dom}(lm) \\
[\![s_1; s_2]\!](m, elm) & = & \mathsf{mlet}\ (m_1, elm_1) = [\![s_1]\!](m, elm)\ \mathsf{in}\ [\![s_2]\!](m_1, elm_1) \\
\hline
[\![P]\!](m) & : & \mathcal{D}(\mathsf{ELMem} \times \mathsf{Val}) \qquad \text{with } m \in \mathsf{Mem} \text{ and } P(\mathbf{a}_1, \ldots, \mathbf{a}_n ::= s; \mathsf{return}\ \mathbf{b} \\
[\![P]\!](m) & = & \mathsf{mlet}\ (m_1, elm_1) = [\![s]\!](m, \emptyset)\ \mathsf{in}\ \mathsf{munit}(elm_1, m_1(\mathbf{b})) \\
\hline
\end{array}
$$

where $m\{x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n\}$ denotes the map m where x_i is updated with v_i for each i in increasing order, and $\uplus$ denotes the disjoint union of partial maps.

Figure 2: Semantics of gadgets and algorithms

4. BASELINE PROBING SECURITY

We first review the basic notion of probabilistic non-interference and state some of its key properties. As usual, we start by introducing a notion of equivalence on memories.

DEFINITION 1. *Let G be a gadget, and let $\mathcal{I} \subseteq \mathbb{I}_G$. Two memories $m, m' \in \mathsf{Mem}$ are $\mathcal{I}$-equivalent, written $m \sim_{\mathcal{I}} m'$, whenever $m(\mathbf{a})^i = m'(\mathbf{a})^i$ for every $\mathbf{a}^i \in \mathcal{I}$.*

Next, we define probabilistic non-interference.

DEFINITION 2. *Let $\mathcal{I} \subseteq \mathbb{I}_G$ and $\mathcal{O} \subseteq \mathbb{O}_G$. A gadget G is* $(\mathcal{I}, \mathcal{O})$-non-interfering *(or $(\mathcal{I}, \mathcal{O})$-NI), iff $[\![G]\!]_{\mathcal{O}}(m) = [\![G]\!]_{\mathcal{O}}(m')$ for every $m, m' \in \mathsf{Mem}$ s.t. $m \sim_{\mathcal{I}} m'$.*

For every gadget G and every position set $\mathcal{O}$, we define the dependency set of $\mathcal{O}$ as $\mathsf{depset}_G(\mathcal{O}) = \bigcap\{\ \mathcal{I} \mid G \text{ is } (\mathcal{I}, \mathcal{O})\text{-NI}\ \}$; thus, $\mathsf{depset}_G(\mathcal{O})$ is the smallest set $\mathcal{I} \subseteq \mathbb{I}_G$ such that G is $(\mathcal{I}, \mathcal{O})$-NI.

LEMMA 1. *Let G be a gadget and $\mathcal{O} \subseteq \mathbb{O}_G$ be a set of positions in G. G is $(\mathsf{depset}_G(\mathcal{O}), \mathcal{O})$-NI.*

We conclude this section by providing an alternative definition of non-interference, in the style of simulation-based security.

LEMMA 2. *A gadget G is $(\mathcal{I}, \mathcal{O})$-NI iff there exists a simulator $\mathbf{Sim} \in (\mathcal{I} \to \mathbb{K}) \to \mathcal{D}(\mathcal{O} \to \mathbb{K})$ such that for every $m \in \mathsf{Mem}$,*

$$[\![G]\!]_{\mathcal{O}}(m) = \mathbf{Sim}(m_{|\mathcal{I}})$$

where $m_{|\mathcal{I}}$ is the restriction of m to elements in $\mathcal{I}$.

This observation is useful to connect the information-flow based formulation of probing security introduced below with the simulation-based formulations of probing security often used by cryptographers. Indeed, the dependency set $\mathsf{depset}_G(\mathcal{O})$ can be interpreted as a set of G's input shares that is sufficient to perfectly simulate the joint distribution of positions in $\mathcal{O}$ to an adversary.

Next we define our baseline notion of probing security, which we call t-non-interference, and state some of its basic properties.

$\mathcal{D}(\mathsf{Val} \times (\mathbb{O}_G \to \mathbb{K}))$. Similarly, one can refine the type of $[\![P]\!]$ to $\mathsf{Mem} \to \mathcal{D}(\mathsf{Val} \times (\mathbb{O}_P \to \mathbb{K}))$.

The notion of t-non-interference is based on the notion of degree of an input set, which we define first. Given an input set $\mathcal{I}$ and an encoding variable $\mathbf{a}$, we define the set $\mathcal{I}_{|\mathbf{a}} \triangleq \mathcal{I} \cap \overline{\mathbf{a}}$ of positions in $\mathcal{I}$ that correspond to shares of $\mathbf{a}$. Further, we define the degree of an input set $\mathcal{I}$ as $\|\mathcal{I}\| \triangleq \max_{\mathbf{a}} |\mathcal{I}_{|\mathbf{a}}|$ (where $|\cdot|$ is the standard notion of cardinality on finite sets). This notion captures the intuition that the adversary should not learn all shares of any single encoding variable, by bounding the information an adversary may learn about any of a gadget's shared inputs through positions probed internally to that gadget.

DEFINITION 3 (PROBING SECURITY). *A gadget G is t-non-interfering (or t-NI) if $\|\mathsf{depset}_G(\mathcal{O})\| \leq |\mathcal{O}|$ for every $\mathcal{O} \subseteq \mathbb{O}_G$ such that $|\mathcal{O}| \leq t$.*

The next lemma establishes that t-NI is already achieved under a weaker cardinality constraint on the dependency set. Variants of Lemma 3 in simulation-based settings appear in [11, 9].

LEMMA 3. *A gadget G is t-NI iff $\|\mathsf{depset}_G(\mathcal{O})\| \leq t$ for every $\mathcal{O} \subseteq \mathbb{O}_G$ s.t. $|\mathcal{O}| \leq t$.*

The notion of t-non-interference extends readily to algorithms. In addition, one can prove that an algorithm is secure iff the gadget obtained by fully inlining the algorithm is secure.

LEMMA 4. *A program P is t-NI iff the gadget $\mathsf{inline}(P)$ obtained by full inlining is t-NI.*

The lemma sheds some intuition on the definition of t-NI for algorithms. However, we emphasize that verifying fully inlined algorithms is a bad strategy; in fact, previous work indicates that this approach does not scale, and that composition results are needed.

5. STRONG NON-INTERFERENCE

We introduce strong non-interference, a reinforcement of probing security based on a finer analysis of cardinality constraints for dependency sets. Informally, strong non-interference distinguishes between internal and output positions, and requires that the dependency set of a position set $\mathcal{O}$ has degree $\leq k$, i.e. contains at most

k shares of each encoding input, where k is the number of *internal* positions in $\mathcal{O}$. Formally, a local variable is an *output position* for G if it appears in the return tuple of G, and an internal position otherwise. Let $\mathcal{O}^{\text{int}}$ (resp. $\mathcal{O}^{\text{ext}}$) denote the subset of internal (resp. output) positions of a set $\mathcal{O}$. Strong t-non-interference requires that the degree of $\mathsf{depset}(\mathcal{O})$ is smaller than $|\mathcal{O}^{\text{int}}|$, rather than $|\mathcal{O}|$. Intuitively, a t-SNI gadget information-theoretically hides dependencies between each of its inputs and its outputs, even in the presence of internal probes. This essential property is what supports compositional reasoning.

DEFINITION 4 (STRONG PROBING SECURITY). *A gadget G is* t-strongly non-interfering *(or t-SNI) if $\|\mathsf{depset}_G(\mathcal{O})\| \leq |\mathcal{O}^{\text{int}}|$ for every position set $\mathcal{O}$ such that $|\mathcal{O}| \leq t$.*

Gadget 4 Mask Refreshing Gadgets

```
0: function RefreshA(a)
1:   c^0 ← a^0
2:   for i = 1 to t do
3:     r ←$ K
4:     c^0 ← c^0 ⊕ r
5:     c^i ← a^i ⊖ r
6:   return c
```
(4a) Addition-Based Mask Refreshing

```
0: function RefreshM(a)
1:   for i = 0 to t do
2:     c^i ← a^i
3:   for i = 0 to t do
4:     for j = i + 1 to t do
5:       r ←$ K
6:       c^i ← c^i ⊕ r
7:       c^j ← c^j ⊖ r
8:   return c
```
(4b) Multiplication-Based Mask Refreshing

Fortunately, many gadgets from the literature achieve strong non-interference (see Table 1 and Section 7). First, we note that gadget RefreshM (Gadget 4b) generalized from Ishai, Sahai and Wagner [25] is t-SNI for all t. (A proof sketch for this proposition is given in Appendix A.)

PROPOSITION 1. RefreshM *(Gadget 4b) is t-SNI.*

On the contrary, the additive refreshing gadget RefreshA (Gadget 4a) achieves NI but fails to achieve SNI. Interestingly, Coron's linear-space variant of Ishai, Sahai and Wagner's multiplication [13, Alg. 6] (Gadget 5a) and the MultLin gadget for securely multiplying linearly dependent inputs [16, Alg. 5] (Gadget 5b) are both strongly non-interfering. The proof of SNI for Gadget 5a is easy to adapt to the more standard quadratic-space multiplication gadget, since they compute the same intermediate values in different orders.

PROPOSITION 2. *The* SecMult *gadget (Gadget 5a) is t-SNI.*

PROPOSITION 3. *The* MultLin *gadget (Gadget 5b) is t-SNI.*

The proofs of Propositions 1, 2 and 3 have been machine-checked using EasyCrypt [6]. We also provide more detailed game-based proof sketches in the full version of this paper.

Strong Non-Interference for Mask Refreshing. We now show how choosing a t-SNI refreshing gadget over a t-NI refreshing gadget critically influences the security of algorithms. Concretely, we provide a separating example, which captures the essence of the flaw in the inversion algorithm of Rivain and Prouff [32]. The example considers two algorithms (Algorithm 6) which compute a cube in $\mathsf{GF}(2^8)$ by squaring and multiplying (using, for illustration purposes, some t-NI gadgets Square and Mult for squaring and multiplication). Both algorithms use a refreshing gadget between the two operations, but they differ in which gadget they use: BadCube

Gadget 5 Some arithmetic gadgets

```
0: function SecMult(a, b)
1:   for i = 0 to t do
2:     c^i ← a^i ⊙ b^i
3:   for i = 0 to t do
4:     for j = i + 1 to t do
5:       r ←$ K
6:       c^i ← c^i ⊖ r
7:       t ← a^i ⊙ b^j
8:       r ← r ⊕ t
9:       t ← a^j ⊙ b^i
10:      r ← r ⊕ t
11:      c^j ← c^j ⊕ r
12:  return c
```
(5a) Masked multiplication [13]

```
0: function MultLin(a)
1:   for i = 0 to t do
2:     c^i ← a^i ⊙ g(a^i)
3:   for i = 0 to t do
4:     for j = i + 1 to t do
5:       r ←$ K
6:       r' ←$ K
7:       c^i ← c^i ⊖ r
8:       t ← a^i ⊗ g(r') ⊕ r
9:       t ← t ⊕ (r' ⊗ g(a^i))
10:      t ← t ⊕ (a^i ⊗ g(a^j ⊖ r')
11:      t ← t⊕((a^j ⊖r')⊗g(a^i))
12:      c^j ← c^j ⊕ t
13:  return c
```
(5b) $x \otimes g(x)$ with linear g [16, Alg. 5]

(Gadget 6a) uses the additive refreshing gagdet RefreshA, which is t-NI but not t-SNI, and Cube (Gadget 6b) uses the RefreshM gadget, which is t-SNI. This simple difference is fundamental for the security of the two algorithms.

Alg. 6 Cubing procedures (with $\mathbb{K} = \mathsf{GF}(2^8)$)

```
function BadCube(x)
  y1 := Square(x)
  y2 := RefreshA(y1)
  z := Mult(x, y2)
  return z
```
(6a) Insecure Cubing

```
function Cube(x)
  y1 := Square(x)
  y2 := RefreshM(y1)
  z := Mult(x, y2)
  return z
```
(6b) Secure Cubing

LEMMA 5 ([16]). BadCube *is not t-NI for any $t \geq 2$.* Cube *is t-NI for all t.*

Coron et al. [16] exhibit proofs for both statements. In Appendix A, we give a compact proof of t-NI for Cube that does not exhaustively consider all $(t+1)$-tuples of positions in Cube. The key argument is that RefreshM being t-SNI essentially renders useless any information on $\mathbf{y}_2$ the adversary may have learned from observing positions in Mult: those do not add any shares of $\mathbf{y}_1$ to the dependency set we compute for RefreshM, and therefore do not influence the shares of $\mathbf{x}$ that appear in the final dependency set for Cube. On the other hand, using a simple t-NI mask refreshing gadget (such as RefreshA) in its place breaks the proof by allowing us to deduce only that each position in the multiplication may depend on 2 shares of $\mathbf{x}$.

In Section 6, we show how the proof of Lemma 5 can be improved and extended into a compositional proof for the (repaired) inversion algorithm of Rivain and Prouff [32], and, in fact, outlines a general mehodology for proving algorithms t-NI or t-SNI.

A Generic Composition Result. Before formalizing and automating this proof process to obtain precise probing security proofs for large circuits, we now give a coarse but simple composition result that illustrates the generality of SNI. Informally, an algorithm is t-NI if all its gadgets verify t-NI and every non-linear usage of an encoding variable is guarded by t-SNI refreshing gadgets. In addition, it shows that processing all inputs, or the output of a t-NI algorithm with a t-SNI gadget (here RefreshM) suffices to make the algorithm t-SNI.

PROPOSITION 4. *An algorithm P is t-SNI provided all its gadgets are t-NI, and all encoding variables are used at most once as argument of a gadget call other than* RefreshM. *Moreover P is t-SNI if it is t-NI and one of the following holds:*

- *its return expression is $\mathbf{b}$ and its last instruction is of the form $\mathbf{b} := \mathsf{RefreshM}(\mathbf{a})$;*
- *its sequence of encoding parameters is $(\mathbf{a}_1, \ldots, \mathbf{a}_n)$, its i^{th} instruction is $\mathbf{b} :=_i \mathsf{RefreshM}(\mathbf{a}_i)$ for $1 \leq i \leq n$, and $\mathbf{a}_i$ is not used anywhere else in the algorithm.*

6. ENFORCING PROBING POLICIES

We first define an expressive assertion language for specifying sets of position sets, and then introduce probing policies, which yield a convenient formalism for defining a large class of information flow policies with cardinality constraints.

DEFINITION 5 (PROBING POLICY).

1. *A* probing assertion *is a pair (Γ, ϕ), where Γ is a map from encoding variables to expressions in the theory of finite sets, and ϕ is a cardinality constraint. Each probing assertion (Γ, ϕ) defines a set of subsets of positions for a fixed algorithm P, denoted by $[\![(\Gamma, \phi)]\!]$. (The syntax and semantics of set expressions and cardinality constraints is explained below.)*
2. *A* probing policy *is a pair of assertions*

$$(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$$

where $(\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$ is the post-assertion *and $(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}})$ is the* pre-assertion.
3. *Algorithm P satisfies the policy $(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$, written $P \models (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$, if for every position set $\mathcal{O} \in [\![(\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})]\!]$, P is $(\mathcal{I}, \mathcal{O})$-NI for some input position set $\mathcal{I} \in [\![(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}})]\!]$.*

The syntax of set expressions and cardinality constraints is given by the following grammar:

$$\begin{array}{lrcl} \text{(set expr.)} & S & := & X \mid \emptyset \mid S \cup S \\ \text{(arith. expr.)} & l & := & |S| \mid |O^\ell| \mid t \mid l + l \\ \text{(cardinality constr.)} & \phi & := & l \leq l \mid \phi \wedge \phi \end{array}$$

The syntax distinguishes between variables X that are drawn from a set $\mathcal{X}$ of names–that we will use to represent sets of shares of an encoding variable, and variables O, annotated with a label ℓ, that are drawn from a disjoint set Ω of names–that we will use to represent sets of internal positions probed in the gadget used at instruction ℓ.

REMARK 1. *Our syntax for set expressions and constraints is a fragment of the (decidable) theory of finite sets with cardinality constraints. It would be possible to include other set-theoretical operations, as in [33, 3]. However, we have found our core fragment sufficient for our purposes.*

The semantics of assertions is defined using the notion of valuation. A valuation μ is a mapping from names in $\mathcal{X}$ and Ω to finite sets, such that $\forall\, X \in \mathcal{X}.\ \mu(X) \subseteq \{0, \ldots, t\}$ and $\forall\, O^\ell \in \Omega.\ \mu(O^\ell) \subseteq \mathbb{O}_{G_\ell}$, where G_ℓ is the gadget called at instruction ℓ. Every valuation μ defines, for every set expression S, a set of share indices $\mu(S) \subseteq \{0, \ldots, t\}$ and for every arithmetic expression l an interpretation $\mu(l) \in \mathbb{N}$, using the intended interpetation of symbols (i.e. $\cup$ is interpreted as set union, $+$ is interpreted as addition, ...).

DEFINITION 6 (INTERPRETATION OF ASSERTIONS).

1. *μ satisfies a cardinality constraint ϕ, written $\mu \models \phi$, if $\mu(l_1) \leq \mu(l_2)$ for every conjunct $l_1 \leq l_2$ of ϕ.*
2. *The interpretation of Γ under μ is the set*

$$\mu(\Gamma) = \bigcup_{\mathbf{a}} \{\mathbf{a}^\imath \mid \imath \in \mu(\Gamma(\mathbf{a}))\} \cup \bigcup_{O} \mu(O)$$

3. *The interpretation of (Γ, ϕ) is the set*

$$[\![(\Gamma, \phi)]\!] = \{\mu(\Gamma) \mid \mu \models \phi\}$$

We now turn to the definition of the type system.

DEFINITION 7. *Algorithm $P(\mathbf{a}_1, \ldots, \mathbf{a}_n) ::= s; \mathsf{return}\ \mathbf{r}$ has type $(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$ if the judgment $\vdash s : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$ is derivable using the typing rules from Figure 3. We denote this fact $\vdash P : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$.*

We briefly comment on the rules. Rule (SEQ) is used for typing the sequential composition of gadget calls and is as expected. The remaining rules are used for interpreting the non-interference properties of gadgets. We now detail them.

Rule (SNI-GADGET) is used for typing calls to a SNI-gadget with an arbitrary post-assertion and a pre-assertion in which the mapping Γ_{out} is updated to reflect the dependencies created by the call, and the constraint is strenghtened with the cardinality constraint imposed by strong non-interference. The rule has a side condition $|O^\ell| + |\Gamma_{\mathsf{out}}(\mathbf{b})| \leq t$ ensuring that the total number of positions whose dependency set by G we are considering is bounded by t, where O^ℓ is the name of the subset of positions that are observed in the current gadget (called at line ℓ), and $\Gamma_{\mathsf{out}}(\mathbf{b})$ is the set of shares of $\mathbf{b}$ the adversary has information about from positions probed in gadgets that use $\mathbf{b}$ later on in the algorithm. This side condition is verified under the condition ϕ_{out}. Note that the variables X_k^ℓ are fresh, and annotated with the label ℓ that identifies the current instruction, and an indice k that identifies the argument. Rule (NI-GADGET) is similar but deals with NI-gadgets, and therefore extends Γ_{in} with correspondingly weaker constraints on the X_k^ℓ.

We now turn to the rule for affine gadgets. Informally, we say that a gadget is affine if it manipulates its input encodings share by share; this includes standard implementations of ring addition, for example, but also of many other functions that are linear in $\mathbb{K}$ (for example, multiplication by a constant–or public–scalar, or shifts in the representation when addition is bitwise). Formally, we say that a gadget G with parameters $(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ is affine iff there exists a family of procedures $f_0, \ldots, f_t$ such that G is an inlining of

$$\begin{array}{l} x_0 \leftarrow f_0(\mathbf{a}_1^0, \ldots, \mathbf{a}_n^0); \ldots; x_t \leftarrow f_t(\mathbf{a}_1^t, \ldots, \mathbf{a}_n^t); \\ \mathbf{return}\ \langle x_0, \ldots, x_t \rangle; \end{array}$$

Thus, one can define a mapping $\eta : \mathcal{O}_G \to \{0, \ldots, t\}$ such that for every position $\pi \in \mathcal{O}_G$, $\eta(\pi) = \imath$ if π occurs in the computation of the $\imath^{\text{th}}$ share (i.e. in the computation of $f_\imath(a_1^\imath, \ldots, a_n^\imath)$). The fine-grained information about dependencies given by this notion of affinity is often critical to proving the probing security of algorithms. Therefore, it is important to capture affinity in our type system. Let $\mathcal{O} = \mathcal{O}^{\text{int}} \uplus \mathcal{O}^{\text{ext}}$ be a position set, split between internal and output positions. The affine property ensures that the joint distribution of $\mathcal{O}$ depends only on input positions in $\eta(\mathcal{O}^{\text{int}} \cup \mathcal{O}^{\text{ext}})$, and furthermore that $|\eta(\mathcal{O}^{\text{int}} \cup \mathcal{O}^{\text{ext}})| = |\eta(\mathcal{O}^{\text{int}})| + |\eta(\mathcal{O}^{\text{ext}})| = |\eta(\mathcal{O}^{\text{int}})| + |\mathcal{O}^{\text{ext}}|$. Rule (AFFINE) interprets this affine property into our type system, using $\Gamma_{\mathsf{out}}(\mathbf{b})$ and a fresh O^ℓ for $\mathcal{O}^{\text{ext}}$ and $\mathcal{O}^{\text{int}}$, respectively, and encoding $\eta(O^\ell)$ into an abstract existential set X^ℓ. The condition $|X^\ell| \leq |O^\ell|$ precisely captures the fact that $|\eta(O)| \leq |O|$ for all O.

DEFINITION 8 (TYPING OF AN ALGORITHM). *Let P be an algorithm defined by $P(\mathbf{a}_1, \ldots, \mathbf{a}_n) ::= s; \mathsf{return}\ \mathbf{r}$.*

$$\frac{\vdash \mathbf{b} := G(\mathbf{a}_1, \ldots, \mathbf{a}_n) : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma, \phi) \qquad \vdash c : (\Gamma, \phi) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})}{\vdash \mathbf{b} := G(\mathbf{a}_1, \ldots, \mathbf{a}_n); c : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})} \quad (\text{SEQ})$$

$$\frac{G \text{ is } t\text{-NI} \qquad \phi_{\mathsf{out}} \Rightarrow |\Gamma_{\mathsf{out}}(\mathbf{b})| + |O^\ell| \leq t \qquad \Gamma_{\mathsf{in}} := \Gamma_{\mathsf{out}}\{\mathbf{b}, \forall\, k.\ \mathbf{a}_k \leftarrow \emptyset, \forall\, k.\ \Gamma_{\mathsf{out}}(\mathbf{a}_k) \cup X_k^\ell\}}{\vdash \mathbf{b} :=_\ell G(\mathbf{a}_1, \ldots, \mathbf{a}_n) : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{out}} \wedge (\bigwedge_{1 \leq k \leq n} |X_k^\ell| \leq |\Gamma_{\mathsf{out}}(\mathbf{b})| + |O^\ell|)) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})} \quad (\text{NI-GADGET})$$

$$\frac{G \text{ is } t\text{-SNI} \qquad \phi_{\mathsf{out}} \Rightarrow |\Gamma_{\mathsf{out}}(\mathbf{b})| + |O^\ell| \leq t \qquad \Gamma_{\mathsf{in}} := \Gamma_{\mathsf{out}}\{\mathbf{b}, \forall\, k.\ \mathbf{a}_k \leftarrow \emptyset, \forall\, k.\ \Gamma_{\mathsf{out}}(\mathbf{a}_k) \cup X_k^\ell\}}{\vdash \mathbf{b} :=_\ell G(\mathbf{a}_1, \ldots, \mathbf{a}_n) : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{out}} \wedge (\bigwedge_{1 \leq k \leq n} |X_k^\ell| \leq |O^\ell|)) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})} \quad (\text{SNI-GADGET})$$

$$\frac{G \text{ is affine} \qquad \Gamma_{\mathsf{in}} := \Gamma_{\mathsf{out}}\{\mathbf{b}, \mathbf{a}_k \leftarrow \emptyset, \Gamma_{\mathsf{out}}(\mathbf{a}_k) \cup \Gamma_{\mathsf{out}}(\mathbf{b}) \cup X^\ell\}}{\vdash \mathbf{b} :=_\ell G(\mathbf{a}_1, \ldots, \mathbf{a}_n) : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{out}} \wedge |X^\ell| \leq |O^\ell|) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})} \quad (\text{AFFINE})$$

where $\Gamma\{\forall\, k.\ v_k \leftarrow \forall\, k.\ e_k\}$ stands for the map Γ where each v_k is updated to map to e_k and all other indices are left untouched.

Figure 3: Typing rules

P is well-typed for NI, written $\vdash_{NI} P$, whenever there exist Γ_{in}, ϕ_{in} such that $\vdash P : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\emptyset, \sum_{1 \leq \ell \leq |P|} |O^\ell| \leq t)$ and, for each $i \in \{1, \ldots, n\}$, $\phi_{\mathsf{in}} \Rightarrow |\Gamma_{\mathsf{in}}(\mathbf{a}_i)| \leq t$.

P is well-typed for SNI, written $\vdash_{SNI} P$, whenever there exist Γ_{in}, ϕ_{in} such that $\vdash P : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow ([\mathbf{r} \leftarrow O], |O| + \sum_{1 \leq \ell \leq |P|} |O^\ell| \leq t)$ and, for each $i \in \{1, \ldots, n\}$, we have $\phi_{\mathsf{in}} \Rightarrow |\Gamma_{\mathsf{in}}(\mathbf{a}_i)| \leq \sum_{1 \leq \ell \leq |P|} |O^\ell|$ (where $[v \leftarrow x]$ is the map that associates x to v and is everywhere else undefined).

When typing for NI, we start from the empty map for Γ_{out} and simply consider any output position observed as if they were internal. However, the same cannot be done when typing for SNI since we need to distinguish clearly between internal positions in one of the O^ℓ, used to type the gadget at instruction ℓ, and output positions in O, initially used as the set of position of the algorithm's return encoding.

PROPOSITION 5 (SOUNDNESS OF THE TYPE SYSTEM). *If $\vdash s : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$ then also*

$$\models P : (\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}}) \Longleftarrow (\Gamma_{\mathsf{out}}, \phi_{\mathsf{out}})$$

If $\vdash_{NI} P$ then P is t-NI
If $\vdash_{SNI} P$ then P is t-SNI

An Example: Rivain and Prouff's inversion algorithm. We now illustrate the type system by describing a typing derivation on Rivain and Prouff's algorithm for computing inversion in $\mathsf{GF}(2^8)$ [32, 16]. An algorithm implementing this operation securely is shown in Figure 4, with some information relevant to its typing derivation. We recall that the function $x \mapsto x^{2^n}$ is linear (for any n) in binary fields and rely on affine gadgets pow2, pow4, and pow16 to compute the corresponding functionalities.

We present the typing derivation in the slightly unusual form of a table, in Figure 4, which shows the code of the inversion algorithm along with the values of Γ_{in} and ϕ_{in} (ϕ_{in} shows only the part of the constraint that is *added* at that program point, not the entire constraint) at each program point. By the sequence rule, these serve as Γ_{out} and ϕ_{out} for the immediately preceding program point. The table also shows the side conditions checked during the application of gadget rules where relevant. It is easier to understand the type-checking process by reading the table from the bottom up.

As per the definition of well-typedness for SNI, we start from a state where the output position set O is associated to the algorithm's return encoding $\mathbf{r}_5$, and where the constraint contains only the global constraint that the whole position set $O \cup \bigcup_\ell O^\ell$ is of cardinality bounded by t. When treating line 9, we know that SecMult is t-SNI and try to apply rule (SNI-GADGET). We check that the number of positions observed in this instance of SecMult is bounded by t (which trivially follows from the global constraint), and construct the new value of $(\Gamma_{\mathsf{in}}, \phi_{\mathsf{in}})$ following the rule: since neither of the call's input encodings are used below, new sets X_1^9 and X_2^9 are associated to the call's inputs and the SNI constraints are added to ϕ_{in}. Applying the rules further until the top of the program is reached, and performing the appropriate set unions in Γ when an encoding variable is used more than once, we observe that the resulting pre-assertion is such that $|\Gamma_{\mathsf{in}}(\mathbf{a})| \leq |O^1| + |O^2| + |O^3| + |O^9| \leq \sum_\ell O^\ell$, and therefore proves that this inversion algorithm is t-SNI.

Finally, one can remark that the instances of SecMult at line 6 and 8 do not in fact need to be t-SNI. As pointed out by Belaïd et al. [9], using a t-NI multiplication gadget at these program points is sufficient to construct a type derivation for SNI.

7. SNI CHECKER FOR GADGETS

We present an automated method for proving that gadgets (or small algorithms, by inlining) are t-SNI at small fixed orders (up to $t = 6$ for ring multiplication). We then give some experimental results.

Verification algorithm. We adapt to t-SNI the algorithmic contributions of Barthe et al. [4] that support the automated verification, on small to medium gadgets and for small orders, of Ishai, Sahai and Wagner's circuit privacy property [25], which is similar to our t-NI. Their work builds on two observations: first, every probabilistic program P taking input x and performing a (statically) bounded number (say q) of uniform samplings over $\mathbb{K}$ is equivalent, in the sense below, to composing a deterministic program $P^\dagger$ taking inputs x and r with random sampling over $\mathbb{K}^q$. Formally, for every x,

$$[\![P]\!](x) = \mathsf{mlet}\ r = \mathcal{U}_{\mathbb{K}^q}\ \mathsf{in}\ [\![P^\dagger]\!]_{\mathcal{O}}(x, r)$$

Second, P satisfies $(\mathcal{I}, \mathcal{O})$-NI iff there exists a function f such that for every x_1, x_2 and r, such that $x_1 \sim_{\mathcal{I}} x_2$

$$[\![P^\dagger]\!]_{\mathcal{O}}(x_1, r) = [\![P^\dagger]\!]_{\mathcal{O}}(x_2, f(x_2, r))$$

and moreover $f(x, \cdot)$ is a bijection for every x. The latter equality can be easily verified for all x and r using standard tools, therefore the key to proving non-interference is to exhibit a suitable function f. Their algorithm proceeds by incrementally defining bijec-

Γ_{in}	ϕ_{in}	Instructions	Side conditions
		function invert($\mathbf{a}$)	
$\mathbf{a} : X_2^3 \cup X_2^9 \cup X_1^2 \cup X^1$	$\lvert X^1\rvert \leq \lvert O^1\rvert$	$\mathbf{z}_1 :=_1$ pow2($\mathbf{a}$)	
$\mathbf{a} : X_2^3;\quad \mathbf{z}_1 : X_2^9 \cup X_1^2$	$\lvert X_1^2\rvert \leq \lvert O^2\rvert$	$\mathbf{z}_2 :=_2$ Refresh($\mathbf{z}_1$)	$\lvert X_1^3\rvert + \lvert O^2\rvert \leq t$
$\mathbf{a} : X_2^3;\quad \mathbf{z}_1 : X_2^9;\quad \mathbf{z}_2 : X_1^3$	$\lvert X_k^3\rvert \leq \lvert O^3\rvert$	$\mathbf{r}_1 :=_3$ SecMult($\mathbf{z}_2, \mathbf{a}$)	$\lvert X_1^6 \cup X_2^8 \cup X_1^5 \cup X^4\rvert + \lvert O^3\rvert \leq t$
$\mathbf{z}_1 : X_2^9;\quad \mathbf{r}_1 : X_1^6 \cup X_2^8 \cup X_1^5 \cup X^4$	$\lvert X^4\rvert \leq \lvert O^4\rvert$	$\mathbf{w}_1 :=_4$ pow4($\mathbf{r}_1$)	
$\mathbf{z}_1 : X_2^9;\quad \mathbf{r}_1 : X_1^6;\quad \mathbf{w}_1 : X_2^8 \cup X_1^5$	$\lvert X_1^5\rvert \leq \lvert O^5\rvert$	$\mathbf{w}_2 :=_5$ Refresh($\mathbf{w}_1$)	$\lvert X_2^6\rvert + \lvert O^5\rvert \leq t$
$\mathbf{z}_1 : X_2^9;\quad \mathbf{r}_1 : X_1^6;\quad \mathbf{w}_1 : X_2^8;\quad \mathbf{w}_2 : X_2^6$	$\lvert X_k^6\rvert \leq \lvert O^6\rvert$	$\mathbf{r}_2 :=_6$ SecMult($\mathbf{r}_1, \mathbf{w}_2$)	$\lvert X_1^8 \cup X^7\rvert + \lvert O^6\rvert \leq t$
$\mathbf{z}_1 : X_2^9;\quad \mathbf{w}_1 : X_2^8;\quad \mathbf{r}_2 : X_1^8 \cup X^7$	$\lvert X^7\rvert \leq \lvert O^7\rvert$	$\mathbf{r}_3 :=_7$ pow16($\mathbf{r}_2$)	
$\mathbf{z}_1 : X_2^9;\quad \mathbf{w}_1 : X_2^8;\quad \mathbf{r}_3 : X_1^8$	$\lvert X_k^8\rvert \leq \lvert O^8\rvert$	$\mathbf{r}_4 :=_8$ SecMult($\mathbf{r}_3, \mathbf{w}_1$)	$\lvert X_1^9\rvert + \lvert O^8\rvert \leq t$
$\mathbf{z}_1 : X_2^9;\quad \mathbf{r}_4 : X_1^9$	$\lvert X_k^9\rvert \leq \lvert O^9\rvert$	$\mathbf{r}_5 :=_9$ SecMult($\mathbf{r}_4, \mathbf{z}_1$)	$\lvert O\rvert + \lvert O^9\rvert \leq t$
$\mathbf{r}_5 : O$	$\lvert O\rvert + \sum_{1\leq\ell\leq 9} \lvert O^\ell\rvert \leq t$	**return** $\mathbf{r}_5$	

Figure 4: $\mathbf{a}^{-1}$ in $\mathsf{GF}(2^8)$

tions $f_1, \ldots, f_n$ satisfying the two conditions above until evenutally $[\![P^\dagger]\!]_{\mathcal{O}}(x, f_n(x, r))$ can be rewritten into an expression that does not depend syntactically on secrets.

However, even with efficient algorithms to prove that a program P is $(\mathcal{I}, \mathcal{O})$-NI for some position set $\mathcal{O}$, proving that P is t-NI remains a complex task: indeed this involves proving $(\mathcal{I}, \mathcal{O})$-NI for all $\mathcal{O}$ with $|\mathcal{O}| \leq t$. Simply enumerating all possible position sets quickly becomes untractable as P and t grow. Therefore, [4] uses the following fact: if P is $(\mathcal{I}, \mathcal{O}')$-NI then it is also $(\mathcal{I}, \mathcal{O})$-NI for all $\mathcal{O} \subseteq \mathcal{O}'$. Hence, checking that P is $(\mathcal{I}, \mathcal{O}')$-NI for some large set $\mathcal{O}'$ is sufficient to prove that P is $(\mathcal{I}, \mathcal{O})$-NI for every $\mathcal{O} \subseteq \mathcal{O}'$, and this using only one proof of non-interference. In particular, they exhibit algorithms that rely on the explicit construction of the bijection f_n to efficiently extend the set $\mathcal{O}$ from which it was constructed into a potentially much larger set $\mathcal{O}'$ for which that bijection still proves $(\mathcal{I}, \mathcal{O}')$-NI. Further, they also exhibit algorithms that rely on such extensions to prove the existence of $\mathcal{I}$ such that $(\mathcal{I}, \mathcal{O})$-NI for all position sets $\mathcal{O}$ much more efficiently than by considering all position sets individually.

We adapt their algorithms by changing the core bijection-finding algorithm in two ways: i. rather than being applied to a modified program that includes the initial uniform sampling of secret encodings, our core algorithm works directly on the gadget description (this is necessary to ensure that we prove t-SNI instead of alternative security notions); and ii. our search for a bijection stops when $[\![P^\dagger]\!]_{\mathcal{O}}(x, f_n(x, r))$ can be simplified into an expression that syntactically depends on at most d shares of the secret (for the desired bound d on $\|\mathcal{I}\|$, that is $d = |\mathcal{O}^{\text{int}}|$ for SNI), rather than stopping when all syntactic dependencies on the secret input have been removed. We note that replacing the bound d from the second point with $d = t$ yields a verification algorithm for t-NI (by Lemma 3). Our full algorithm is given in the long version [5].

Evaluation. We evaluate the performance of our SNI verifier on some medium and small gadgets: SecMult**,** Coron's linear-memory ring multiplication algorithm [13, Alg. 6]; MultLin**,** Coron et al.'s algorithm for the computation of functionalities of the form $\mathbf{x} \odot g(\mathbf{x})$ for some linear g [16, Alg. 5]; Add**,** the standard affine gadget for the addition of two encodings; RefreshA**,** the weakly secure mask refreshing algorithm from Rivain and Prouff [32]; RefreshIter^k**,** the iterated additive refresh proposed by Coron [13, Alg. 4] for supporting more efficient composition in his full model (we make explicit the number of iterations k); WeakMult**,** the generic reduced-randomness multiplication algorithm proposed by Belaïd et al. [9]. Table 1 sums up our findings and some verification statistics.

8. MASKING TRANSFORMATION

As a proof of concept, we implement our type system for a comfortable subset of C that includes basic operators, static for loops, table lookups at public indices, and mutable secret state, and extended with libraries that implement core gadgets for some choices of $\mathbb{K}$. Moreover, we define a source-to-source *certifying masking transformation*, which takes an unprotected program and returns a masked algorithm accepted by our type system, selectively inserting refreshing gadgets as required for typing to succeed. We note that the transformation itself need not be trusted, since its result is the final program on which typing is performed.

Furthermore, the choice of C as a supporting language is for convenience, since many of the algorithms we consider have reference implementations written in C. In particular, we do not claim that compiling and executing the C programs produced by our masking transformation will automatically yield secure executables: our verification results are on *algorithms* described in the C language rather than on C programs in general. Making use of these verification results in practice still requires to take into account details not taken into account in the probing model. Although an important problem, this is out of the scope of this paper and a research area on its own: for example Balasch et al. [2] consider some of the issues involved in securely implementing probing secure algorithms.

8.1 Implementation

We now give an overview of the different passes performed by our masking transformation. The input programs use explicit typing annotations to distinguish public variables (for example, public inputs, or public loop indices) from sensitive or secret variables that must be encoded. We call public type any type outside of those used for denoting variables that must be encoded.

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 encoding gadgets (that set one share to the public value and all other shares to 0).

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,

Gadget	Order 1		Order 2		Order 3		Order 4		Order 5		Order 6	
	1-SNI	Time	2-SNI	Time	3-SNI	Time	4-SNI	Time	5-SNI	Time	6-SNI	Time
SecMult	✓	0.07s	✓	0.08s	✓	0.09s	✓	0.86s	✓	36.40s	✓	37min
MultLin	✓	0.07s	✓	0.08s	✓	0.15s	✓	1.19s	✓	54.13s	✓	48min
RefreshA	✓	0.07s	✗	0.08s	✗	0.08s	–	–	–	–	–	–
RefreshIter[2]	✓	0.08s	✓	0.08s	✓	0.08s	✓	0.08s	✓	0.13s	✗	.20s
RefreshIter[3]	–	–	✓	0.09s	✓	0.08s	✓	0.09s	✓	0.14s	✓	.54s
WeakMult	✓	0.07s	✗	0.07s	✗	0.09s	–	–	–	–	–	–

Table 1: Experimental Results for the SNI Verifier

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. Since the encoding replaces scalar-typed variables (passed by value) with array-typed variables (passed by reference), it is also necessary to slightly transform the program to ensure the correctness of the resulting program. In addition, we also transform the input program into a form that more closely follows the abstract language from Figure 1, which makes it easier to type-check.

Type Inference and Refresh Insertion. This is the core of our transformation. We implement a type inference algorithm for the type system of Section 6. The algorithm simplifies policies on the fly, supports inferred types on sub-algorithms as gadget-invocation types, and fails when the simplified policy is inconsistent. Failure arises exactly when a refreshing operation is needed. At the cost of tracking some more information and reinforcing the typing constraint on sub-algorithms, we use this observation to automatically insert Refresh gadgets where required. When type inference 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 transformation keeps track of typing information for each program point already traversed and simply rewinds the typing to the program point immediately after the modification.

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, and to remove initialization code on normalized gadgets. Interestingly, our transformation produces a (set of) C files that is parameterized by the masking order t. Producing executable versions of that algorithm at a particular order, for example to evaluate its performance, is as easy as defining a pre-processor macro at compile-time.

8.2 Practical Evaluation

To test the effectiveness of our transformation, we apply it to different algorithms, generating equivalent masked algorithms at various orders. We apply our transformation to the following programs: **AES ($\odot$),** a full computation (10 rounds including key schedule) of AES-128 masked using the multiplication gadget, and implemented in $\mathsf{GF}(2^8)$; **AES ($x \odot g(x)$),** a full computation (10 rounds including key schedule) 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 $\mathsf{GF}(2^{64})$; **Simon,** a block of Simon(128,128), implemented in $\mathsf{GF}(2^{64})$; **Speck,** a block of Speck(128,128), implemented in $\mathsf{GF}(2)^{64}$, and using one of the following modular addition algorithms; **AddLin,** Coron, Großschädl and Vadnala's algorithm [15] for the computation of modular addition on boolean-masked variables (in $\mathsf{GF}(2)^{64}$); **AddLog,** Coron et al.'s improved algorithm [14] for the computation of modular addition on boolean-masked variables (in $\mathsf{GF}(2)^{64}$). We first discuss the performance of our verifier and the verification results before discussing the practical significance, in terms of time, memory and randomness complexity of our masking transformation. Finally, we discuss examples on which our tool implementation could be improved.

Verification Performance and Results. Table 2 shows resource usage statistics for generating the masked algorithms (at any order) from unprotected implementations of each algorithm. The table shows the number of mask refreshing operations inserted in the program[5], the compilation time, and the memory consumption. For Keccak, we show two separate sets of figures: the first, marked "no refresh", is produced by running our algorithm transformer on a bare implementation of the algorithm; the second, marked "refresh in χ", is produced by running our tool on an annotated implementation, where a mask refreshing operation is manually inserted in the χ function and the tool used for verification only. We discuss discrepancies between the numbers on these two lines in Section 9, and consider the "refresh in χ" set of statistics in all discussions until then. We first note the significant improvements these results represent over the state of the art in formal verification for probing security. Indeed, our closest competitor [4] report the verification of all 10 rounds of AES (including key schedule) at order 1 in 10 minutes, and could not verify all 10 rounds for higher orders. In contrast, our tool verifies the probing security of Rivain and Prouff's algorithm [32] as fixed by Coron et al. [16] *at all orders* in less than a second.[6] Further, we note that the masked algorithms our transformation produce for modular addition are the first such algorithms known to be t-probing secure using only $t+1$ shares. Indeed, the original proofs [15, 14] rely on the ISW framework and make use of $2t+1$ shares to obtain t-probing security. We further note that Coron, Großschädl and Vadnala's algorithm [15] does not require the insertion of mask refreshing operations, and is thus t-probing secure with $t+1$ shares as it was originally described. Finally, we note that, to the best of our knowledge, the results obtained on Keccak, Simon and Speck constitute the first generic higher-order masking schemes for these algorithms.

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

[6]This excludes the once-and-forall cost of proving the security of core gadgets.

Algorithm	# Refresh	Time	Mem.
AES ($\odot$)	2 per round	0.09s	4MB
AES ($x \odot g(x)$)	0	0.05s	4MB
AddLin	0	0.01s	4MB
AddLog	$\log_2(k) - 1$	0.01s	4MB
Keccak (no refresh)	1 per round	$\sim$20min	23GB
Keccak (refresh in χ)	0	18.20s	456MB
Simon	67 per round	0.38s	15MB
Speck (AddLin)	61 per round	0.35s	38MB
Speck (AddLog)	66 per round	0.21s	8MB

Table 2: Resource usage during masking and verification

Performance of Masked Algorithms. Table 3 reports the time taken to execute the resulting programs 10,000 times at various orders on an Intel(R) Xeon(R) CPU E5-2667 0 @ 2.90GHz with 64GB of memory running Linux (Fedora). As an additional test to assess the performance of the generated algorithms at very high orders, we masked an AES computation at order 100: computation took $\sim$0.11 seconds per block. For AES and Speck, the figures shown in the "unmasked" column are execution times for the input to our transformation: a table-based implementation of AES or an implementation of Speck that uses machine arithmetic, rather than Coron, Großschädl and Vadnala's algorithm would be much faster, but cannot be masked directly using our transformation. Although these observations do highlight the cost of security, we note that using RefreshA when masking the AES SBox does not incur a significant timing gain for any of the masking orders we tested ($t \leq 20$). However, the randomness cost is greatly reduced, which may be significant in hardware or embedded software settings. Further research in reducing the randomness cost of SNI mask refreshing, or of other gadgets, serves to make security less costly [9, 1, 7]. We also confirm the 15% timing improvements reported by Coron et al. [16] when implementing the AES SBox using their gadget for computing $\mathbf{x} \odot g(\mathbf{x})$.

We now look more closely at statistics for the modular addition algorithms AddLin and AddLog and their effects on the performance of masked algorithms for Speck. We first note that proving AddLog t-NI requires the addition of a mask refreshing gadget, whereas AddLin does not. Despite this additional cost, however, AddLog is better than AddLin when word size k grows, since it saves $k - \log(k)$ multiplications and replaces them with a single mask refreshing operation. These performance gains on modular addition become overwhelming when seen in the context of a masked algorithm for Speck, which computes one 64-bit modular addition per round. It would be interesting to consider using our transformer to produce masked algorithms for other efficient circuits for modular addition [27] and measure their performance impact in terms of randomness, time and memory when masked.

9. DISCUSSIONS AND RELATED WORK

Here, we further discuss the relation between the definitions and results reported here and existing and future work in theoretical and practical cryptography. Our discussions focus mainly on: i. adversary and leakage models; ii. compositional security notions; iii. theoretical and practical masking transformations; and iv. limitations of our definitions and tools.

Adversary and Leakage Models for Masking. We have considered security in the probing model of Ishai, Sahai and Wagner [25], which is particularly well-suited to automated analysis due to its tight relation to probabilistic non-interference. In particular, our notion of t-NI is equivalent to the notions of *t-probing security* and *perfect t-probing security* used by Carlet et al. [11] and others [32, 16].

Despite its broad usage in the literature, the practical relevance of the probing model is not immediately obvious: in practice, side-channel adversaries observe *leakage traces*, which contain noisy information about all intermediate computations, rather than precise information about some. This threat model is much more closely captured by the *noisy leakage model*, first introduced by Chari et al. [12] and extended by Prouff and Rivain [31]. The noisy leakage model is much more complex and makes security proofs on masked algorithms significantly more involved, and much harder to verify.

Duc, Dziembowski and Faust [18] show that proving probing security allows one to estimate the practical (noisy leakage) security of a masked algorithm. While Duc, Faust and Standaert [19] empirically show that some of the factors of Duc *et al.*'s bound [18] are likely proof artefacts, the remainder of the bound, and in particular a factor that includes the size of the circuit, seems to be tight. Intuitively, Duc *et al.* [19] essentially show that the probing security order gives an indication of the smallest order moment of the distribution over leakage traces that contains information about the secret, whereas the size of the circuit the adversary can probe is an indicator of how easy it is to evaluate higher-order moments.

Composition, and Region and Stateful Probing. This observation makes clear the importance of also considering more powerful probing adversaries that may place t probes in each of some (pre-determined) *regions* of an algorithm (the *t-region probing model*). For example, each core gadget (field operations and mask refreshing operation) could be marked off as a separate region (as in [18]). More recently, and in work contemporary with that presented here, Andrychowicz, Dziembowski and Faust [1] consider a more general notion of region whose size must be linear in the security parameter (and masking order), and exhibit a mask refreshing gadget that is linear in size and fulfills, in the probing model, the *reconstructibility* and *re-randomization* properties from Faust et al. [22]. We now discuss the implications of reconstructibility and re-randomization, and their relation to our notion of SNI, based on the similarity of Prop. 4 with Ishai *et al.*'s remark on "Re-randomized outputs" [25], before discussing the applicability of SNI to security in the region and stateful probing models [25].

Intuitively, a gadget is t-reconstructible whenever any t of its positions can be simulated using only its (shared) inputs and outputs, and a gadget is re-randomizing whenever its output encoding is uniform and t-wise independent even if its input encoding is completely known. Our SNI notions combines both considerations. Formulating it in similar terms, a gadget is t-SNI whenever any t of its positions can be simulated using only its (shared) inputs, and if its output encoding is uniform and $(t-d)$-wise independent even if d shares of each of its inputs are known (for all d such that $0 \leq d < t$). Expressed in this way, it is clear that SNI is slightly weaker than "reconstructible and re-randomizable" in the probing model. This allows us to automatically verify that a gadget is SNI for some fixed t, whereas reconstructibility and re-randomization are more complex. In addition, the ability to combine the use of SNI and weaker (NI or affine) gadgets in a fine-grained way allows us to more precisely verify the security of large algorithms in models where the adversary can place t probes in the entire algorithm. We leave a formal investigation of the relation between SNI and "reconstructibility and re-randomization" as future work.

Based on reconstructibility and re-randomization, Faust et al. [22, 1] prove elegant and powerful composition results that in fact apply

Algorithm	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 ($x \odot g(x)$)	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 (AddLin)	0.022s	4.361s	10.281s	20.053s	47.389s	231.423s	357.153s	603.261s
Speck (AddLog)	0.022s	0.529s	1.231s	2.258s	5.621	19.991s	42.032	72.358s

Table 3: Time taken by 10,000 executions of each program at various masking orders

in the more powerful region probing and stateful probing models [25], where the adversary may (adaptively) place t probes in each region (or in each subsequent iteration) of the algorithm. It is worth noting that our SNI notion also enables composition in these two models: indeed, it is easy to see that any two $2t$-SNI algorithms (our regions) can be composed securely when the adversary can place t probes in each of them. Further, our composition techniques also support elegant constructions that support compositional security proofs in the region and stateful probing models without doubling the number of shares computations are carried out on (instead, simply doubling the number of shares at region boundaries). We give details of these *robust composition* results in the full version. Depending on the size of regions that are considered, these robust composition results may bring significant performance gains in terms of randomness and time complexity.

Finally, our notion of SNI and the automated verification techniques presented allow the efficient, precise and automated verification of t-SNI inside each region, an issue which is not addressed by the works of Faust et al. [22, 1].

Existing Masking Transformations. Ishai, Sahai and Wagner [25] and others [18, 1] also propose simple masking transformations that turn unprotected algorithms (or boolean or arithmetic circuits) into protected masked algorithms. Ishai, Sahai and Wagner [25] forgo the use of mask refreshing gadgets by doubling the number of shares on which masked computations occur–with a quadratic impact on performance and randomness complexity. Faust et al. [18, 1] rely on making sure that all gadgets used in the masked algorithm are reconstructible and re-randomizing. This guarantees security in a stronger probing model, but incurs an even greater loss of performance. By contrast, our transformation attempts to decide whether a mask refreshing operation is required to ensure security in the probing model, and our core contributions (the notion of SNI and the type-checker) do support composition in stronger probing models, whilst still allowing the proofs of security within regions to be handled precisely.

Coron [13] proposes schemes for masking lookups at secret or sensitive indices in public tables. We have not investigated whether or not the proposed algorithms are SNI or simply NI, and whether or not establishing these properties can be done by adapting our type-system or if it should be done in a different way (either as a direct proof or using the checker from Section 7). We note in passing that part of the result by Coron [13], namely that using $\mathsf{RefreshIter}_{2t}^{2t+1}$ between each query to the masked S-box supports security in the stateful probing model is subsumed and improved by the robust composition results described in the full version.

The security analysis of masking schemes in the t-probing model is connected to techniques from multi-party computation, exploited in parallel lines of research by threshold implementations [29, 10]. In particular, higher-order threshold implementations are exposed to similar security issues due to composition, although they offer additional protection against practical considerations not captured in standard probing models, namely *glitches*. We believe that the results discussed here are in fact applicable to the compositional security analysis of threshold implementations but leave a formal investigation of these links as future work.

Refining SNI. We now discuss some limitations of our current implementation, and leads for future theoretical work that may yield significant practical improvements.

Alg. 7 Semi Public Modular Addition in $\mathsf{GF}(2)^k$

function $\mathsf{AddPub}(\mathbf{x}, y)$
$\mathbf{w} := \mathbf{x} \odot y$
$\mathbf{a} := \mathbf{x} \oplus y$
$\mathbf{u} := \mathbf{w} \ll 1$
for $i = 2$ **to** $k - 1$ **do**
$\mathbf{a}' := \mathsf{RefreshM}(\mathbf{a})$
$\mathbf{ua} := \mathbf{u} \odot \mathbf{a}'$
$\mathbf{u} := \mathbf{ua} \oplus \mathbf{w}$
$\mathbf{u} := \mathbf{u} \ll 1$
$\mathbf{z} := \mathbf{a} \oplus \mathbf{u}$
return z

(7a) Masked algorithm produced by our tool

function $\mathsf{AddPub}(\mathbf{x}, y)$
$\mathbf{w} := \mathbf{x} \odot y$
$\mathbf{a} := \mathbf{x} \oplus y$
$\mathbf{w} := \mathsf{RefreshM}(\mathbf{w})$
$\mathbf{u} := \mathbf{w} \ll 1$
for $i = 2$ **to** $k - 1$ **do**
$\mathbf{ua} := \mathbf{u} \odot \mathbf{a}$
$\mathbf{u} := \mathbf{ua} \oplus \mathbf{w}$
$\mathbf{u} := \mathbf{u} \ll 1$
$\mathbf{z} := \mathbf{a} \oplus \mathbf{u}$
return z

(7b) Masked algorithm produced by hand

The first point we wish to discuss is the case of Keccak, for which algorithm transformation is prohibitively expensive. This issue is due to our handling of static for loops: indeed, our tool unrolls them to perform type-checking and rolls them back up afterwards if possible (otherwise leaving them unrolled in the final algorithm). For smaller algorithms, this is not a problem, but unrolling all 24 rounds of Keccak-f, along with all the loops internal to each iteration, yields a very large program that is then backtracked over each time a mask refreshing operation is inserted. Refining our non-interference notions to multi-output gadgets and algorithms would allow us to significantly improve our tool's handling of loops and high-level composition, whilst gaining a better understanding of probing security in such scenarios. This improved understanding may in turn help inform the design of primitives that are easier to protect against higher-order probing.

Second, we discuss our greedy policy for the insertion of mask refreshing algorithms. In our experiments, we consider a version of the linear-time modular addition algorithm [15] whose second argument is a public (non-shared) value (for example, a round counter, as in Speck). We show its code, as produced by our masking transformer, in Gadget 7a, and display a hand-masked variant in Gadget 7b, slightly abusing notations by denoting simple gadgets with the symbol typically used for their unprotected versions. Notice that the variable $\mathbf{w}$ is used once per loop iteration, and that our tool refreshes each of them, while it is sufficient to mask only the first one. Improving our gadget selection algorithm to detect and implement this optimization—and others—would be an interesting avenue for

future work, that could help improve our understanding of the effect on security of compiler optimizations.

Acknowledgements. The work presented here was supported by projects S2013/ICE-2731 N-GREENS Software-CM, ANR-10-SEGI-015 PRINCE and ANR-14-CE28-0015 BRUTUS, and ONR Grants N000141210914 and N000141512750, as well as FP7 Marie Curie Actions-COFUND 291803.

10. REFERENCES

[1] Marcin Andrychowicz, Stefan Dziembowski, and Sebastian Faust. Circuit compilers with $O(1/\log(n))$ leakage rate. In *EUROCRYPT 2016*, LNCS, pages 586–615. Springer, Heidelberg, 2016.

[2] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Proceedings of the Smart Card Research and Advanced Application Conference (CARDIS)*, volume 8968 of *LNCS*, pages 64–81. Springer, Heidelberg, November 2014.

[3] Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. In *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *LNCS*, pages 82–98, June 2016.

[4] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, April 2015.

[5] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. Cryptology ePrint Archive, Report 2015/506, 2015. http://eprint.iacr.org/2015/506.

[6] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, 2013.

[7] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *CHES 2016*, LNCS, pages 23–29. Springer, Heidelberg, 2016.

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

[9] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *EUROCRYPT 2016*, LNCS, pages 616–648. Springer, Heidelberg, 2016.

[10] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 326–343. Springer, Heidelberg, December 2014.

[11] Claude Carlet, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Algebraic decomposition for probing security. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 742–763. Springer, Heidelberg, August 2015.

[12] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[13] 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, Heidelberg, May 2014.

[14] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Heidelberg, March 2015.

[15] 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, Heidelberg, September 2014.

[16] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, Heidelberg, March 2014.

[17] Jean-Sébastien Coron, Aurélien Greuet, Emmanuel Prouff, and Rina Zeitoun. Faster evaluation of sboxes via common shares. In *CHES 2016*, LNCS, pages 498–514. Springer, Heidelberg, 2016.

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

[19] 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, Heidelberg, April 2015.

[20] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *Proceedings of the 26th International Conference on Computer Aided Verification.*, pages 114–130, 2014.

[21] Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 62–77, 2014.

[22] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 135–156. Springer, Heidelberg, May 2010.

[23] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, Heidelberg, August 1999.

[24] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? Cryptology ePrint Archive, Report 2016/264, 2016. http://eprint.iacr.org/.

[25] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[26] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

[27] Thomas Walker Lynch. Binary adders, 1996.

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

[29] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, April 2011.

[30] Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Computer Security Foundations Symposium*, pages 75–89. IEEE Computer Society, 2015.

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

[32] 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, Heidelberg, August 2010.

[33] Calogero G. Zarba. Combining sets with cardinals. *Journal of Automated Reasoning*, 34(1):1–29, 2005.

[34] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In Keith Marzullo and M. Satyanarayanan, editors, *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 1–14. ACM, 2001.

Appendices

A. PROOFS

This appendix lists proofs omitted from the main body of the paper, and whose text may add to the reader's comprehension.

PROOF SKETCH FOR PROPOSITION 1. Leveraging the equivalence between simulation and non-interference (Lemma 2), we prove t-SNI by constructing a simulator that uses at most $|\mathcal{O}^{\text{int}}|$ shares of the gadget's input to perfectly simulate the joint distribution of any position set $\mathcal{O}$ such that $|\mathcal{O}| \leq t$. The constructed simulator is very similar to those previously used in proofs of t-NI.

Let $\mathcal{O}$ be a set of positions such that $|\mathcal{O}| \leq t$, and let $d_1 = |\mathcal{O}^{\text{int}}|$ and $d_2 = |\mathcal{O}^{\text{ext}}|$. Note that $d_1 + d_2 \leq t$. Our goals are: i. to find an input set $\mathcal{I}$ such that $\|\mathcal{I}\| \leq d_1$, ii. to construct a perfect simulator that uses only input shares $\mathbf{a}^i \in \mathcal{I}$.

First, we identify which variables are internal (and therefore will be considered in $\mathcal{O}^{\text{int}}$) and which are outputs (in $\mathcal{O}^{\text{ext}}$). Internals are the $\mathbf{a}^i$, the $r_{i,j}$ (the value of r sampled 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 of the second loop. Outputs are the final values of $\mathbf{c}^i$ (i.e. $c_{i,t}$). Then, we define $\mathcal{I}$ as follows: for each position among $\mathbf{a}^i$, $r_{i,j}$ and $c_{i,j}$ (with $j < t$) we add share $\mathbf{a}^i$ to $\mathcal{I}$. It is clear that $\mathcal{I}$ contains at most d_1 positions, since each internal position adds at most one position to $\mathcal{I}$. We now construct the simulator. For clarity, observe that the RefreshM algorithm can be represented using the following matrix, observing that $c_{i,j}$ is the partial sum of the first $j+2$ elements of line i.

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

For each $\imath_0$ such that $\mathbf{a}^{\imath_0} \in \mathcal{I}$ (that is, for each line $\imath_0$ of the matrix that contains at least one observed internal value), $\mathbf{a}^{\imath_0}$ is provided to the simulator (by definition of $\mathcal{I}$). Thus, the simulator can sample all $r_{\imath_0,j}$ and compute all partial sums $c_{\imath_0,j}$ and the $\imath_0^{\text{th}}$ output normally. At this point, all values (all internals and all outputs) on lines indexed by an $\imath$ with $\mathbf{a}^\imath \in \mathcal{I}$ follow the same distribution as they would in the real computation and are therefore perfectly simulated.

It remains to simulate output shares $\mathbf{c}^\jmath$ when $\mathbf{a}^\jmath \notin \mathcal{I}$. Remark that simulating the $\imath^{\text{th}}$ line as above also necessarily fixes the value of all random variables appearing in the $\imath^{\text{th}}$ column. After internal positions are simulated, at most d_1 lines of the matrix are fully filled. Therefore, each line $\jmath$ with $\mathbf{a}^\jmath \notin \mathcal{I}$ contains at least $t - d_1 \geq d_2$ holes corresponding to random values that have not yet been fixed. For each of the output position made on one such line $\jmath$, we can therefore pick a different $r_{\jmath,k}$ that we choose so $\mathbf{c}^\jmath$ can be simulated by a freshly sampled uniform value. □

PROOF SKETCH FOR LEMMA 5. [16] exhibit proofs for both statements. We now sketch a proof of t-NI for Cube that does not exhaustively consider all $(t+1)$-tuples of positions in Cube, emphasizing the critical use of strong non-interference for the refreshing gadget. Recall that our goal is to upper-bound $\|\mathsf{depset}_{\mathsf{Cube}}(\mathcal{O})\|$ for all $\mathcal{O} \subseteq \mathbb{O}_{\mathsf{Cube}}$ such that $|\mathcal{O}| \leq t$. Given such a set, we first partition it as $\mathcal{O} \triangleq \mathcal{O}_{\mathsf{M}} \uplus \mathcal{O}_{\mathsf{R}} \uplus \mathcal{O}_{\mathsf{S}}$ following gadget boundaries (recall that positions in algorithms include the label of the gadget invocation they occur in). First, we consider the dependency set $\mathcal{I}_{\mathsf{M}} \triangleq \mathsf{depset}_{\mathsf{Mult}}(\mathcal{O}_{\mathsf{M}})$ of $\mathcal{O}_{\mathsf{M}}$ by Mult. We know $|\mathcal{O}_{\mathsf{M}}| \leq |\mathcal{O}| \leq t$, and by t-NI of Mult, we deduce that $\|\mathcal{I}_{\mathsf{M}}\| \leq |\mathcal{O}_{\mathsf{M}}|$. Considering now the invocation of RefreshM, we must establish cardinality properties of the dependency set $\mathcal{I}_{\mathsf{R}} \triangleq \mathsf{depset}_{\mathsf{RefreshM}}(\mathcal{O}_{\mathsf{R}} \cup \mathcal{I}_{\mathsf{M}|\mathbf{y}_2})$ of those direct *internal* positions observed by the adversary in RefreshM, *jointly* with those *output* positions she may have learned information about through positions probed in later parts of the circuit (here, in Mult). From previous inequalities, we know that $|\mathcal{I}_{\mathsf{M}|\mathbf{y}_2}| \leq \|\mathcal{I}_{\mathsf{M}}\| \leq |\mathcal{O}_{\mathsf{M}}|$, and thus we have $|\mathcal{O}_{\mathsf{R}} \cup \mathcal{I}_{\mathsf{M}|\mathbf{y}_2}| \leq t$. By t-SNI of RefreshM, we thus have $\|\mathcal{I}_{\mathsf{R}}\| \leq |\mathcal{O}_{\mathsf{R}}|$ (since positions in $\mathcal{I}_{\mathsf{M}|\mathbf{y}_2}$ are external to RefreshM). Finally, we consider the dependency set $\mathcal{I}_{\mathsf{S}} \triangleq \mathsf{depset}_{\mathsf{Square}}(\mathcal{O}_{\mathsf{S}} \cup \mathcal{I}_{\mathsf{R}|\mathbf{y}_1})$ of direct internal positions observed by the adversary in Square, jointly with those output positions she may have learned information about through positions probed in later parts of the circuit (here in RefreshM and Mult, propagated through the single use of $\mathbf{y}_1$ in the invocation of RefreshM). Since we have $|\mathcal{O}_{\mathsf{S}} \cup \mathcal{I}_{\mathsf{R}|\mathbf{y}_1}| \leq |\mathcal{O}_{\mathsf{S}}| + \|\mathcal{I}_{\mathsf{R}}\| \leq |\mathcal{O}_{\mathsf{S}}| + |\mathcal{O}_{\mathsf{R}}| \leq t$, and since Square is t-NI, we conclude that $\|\mathcal{I}_{\mathsf{S}}\| \leq |\mathcal{O}_{\mathsf{S}}| + |\mathcal{O}_{\mathsf{R}}|$. Overall, we have $\mathsf{depset}_{\mathsf{Cube}}(\mathcal{O}) \subseteq \mathcal{I}_{\mathsf{S}|\mathbf{x}} \cup \mathcal{I}_{\mathsf{M}|\mathbf{x}}$, and we can conclude (using some of the intermediate inequalities from above), that $\|\mathsf{depset}_{\mathsf{Cube}}(\mathcal{O})\| \leq |\mathcal{O}_{\mathsf{S}}| + |\mathcal{O}_{\mathsf{R}}| + |\mathcal{O}_{\mathsf{M}}| \leq t$. □