Verified Proofs of Higher-Order Masking

Gilles Barthe[1(✉)], Sonia Belaïd[2], François Dupressoir[1], Pierre-Alain Fouque[3], Benjamin Grégoire[4], and Pierre-Yves Strub[1]

[1] IMDEA Software Institute, Madrid, Spain
{gilles.barthe,francois.dupressoir,pierre-yves.strub}@imdea.org
[2] École normale supérieure and Thales Communications and Security, Paris and Gennevilliers, France
sonia.belaid@ens.fr
[3] Université de Rennes 1 and Institut universitaire de France, Rennes, France
pierre-alain.fouque@ens.fr
[4] INRIA, Sophia-Antipolis, France
benjamin.gregoire@inria.fr

Abstract. In this paper, we study the problem of automatically verifying higher-order masking countermeasures. This problem is important in practice, since weaknesses have been discovered in schemes that were thought secure, but is inherently exponential: for t-order masking, it involves proving that every subset of t intermediate variables is distributed independently of the secrets. Some tools have been proposed to help cryptographers check their proofs, but are often limited in scope.

We propose a new method, based on program verification techniques, to check the independence of sets of intermediate variables from some secrets. Our new language-based characterization of the problem also allows us to design and implement several algorithms that greatly reduce the number of sets of variables that need to be considered to prove this independence property on *all* valid adversary observations. The result of these algorithms is either a proof of security or a set of observations on which the independence property cannot be proved. We focus on AES implementations to check the validity of our algorithms. We also confirm the tool's ability to give useful information when proofs fail, by rediscovering existing attacks and discovering new ones.

Keywords: Higher-order masking · Automatic tools · EasyCrypt

1 Introduction

Most widely used cryptographic algorithms are assumed to be secure in the *black-box* model, that is when the adversary is only given access to the inputs and outputs of the algorithm. However, this model does not fit the reality of embedded devices. In practice, an attacker can observe the physical leakage of a device in order to mount *side-channel attacks*. These attacks exploit the dependence between secret values used in the computation and the physical

E. Oswald and M. Fischlin (Eds.): EUROCRYPT 2015, Part I, LNCS 9056, pp. 457–485, 2015.
DOI: 10.1007/978-3-662-46800-5_18

leakage inherent to the physical implementation and execution (for example, timing, power consumption or electromagnetic radiations). Such attacks are very efficient in practice, with a recent attack recovering a full AES key using a single power trace [34]. Further, *differential attacks* can be mounted that exploit similar dependencies between *sensitive* values, that depend on both secret inputs and adversarially-controlled public inputs, to speed up the rate at which information on the secrets is gathered. Differential Power Analysis [22] (DPA), in particular, is a very effective class of attacks.

Masking. In order to thwart Differential Power Analysis, the community have proposed many countermeasures but *masking* remains most widely used. Masking makes use of a secret-sharing scheme to split each secret or sensitive variable into $(t+1)$ shares such that the joint distribution of any subset of at most t shares is independent of the secret, but the knowledge of all $(t+1)$ shares allows for the efficient recovery of the secret. The computation itself is then masked as well, replacing basic operations on, say, bytes with complex operations on $(t+1)$ bytes. Intuitively, an implementation that is split over $(t+1)$ shares should be able to resist the leakage of t of its intermediate variables (t is then usually called the *masking order*). Most of the implementations were then masked at order 1 to prevent an adversary from recovering secrets using a single observation. However, even higher-order attacks, where t is greater than 1 have been conducted in practice [26,28] and need to be protected against. Many masked implementations have been proposed to protect AES or its non-linear component, the S-box (for example, [10,27,30,31,33]), among which some are also proved secure. Checking first-order masking schemes is a relatively routine task since it is sufficient to check that each intermediate variable carries a distribution that is independent from the secret. However, manually checking higher-order masked implementations is a more difficult and error-prone task. As a consequence, many published schemes were later shown to be insecure, such as those presented by [33] and [31], which were later broken in [12] and [13]. In this paper, we address this issue by developing automated methods to verify the security of algorithms masked at higher orders.

Adversary Models. The first step towards formally reasoning about the security of masked algorithms is to define a leakage model that formally captures the information that is leaked to the adversary. For this purpose, Chari et al. [11] perform the first formal security analysis of masking, by showing that the number of queries needed to recover a sensitive bit in a *noisy leakage model* is at least exponential in the masking order. In this model, the adversary gets leaked values sampled according to a Gaussian distribution centered around the actual value of the wire. This model is later extended by Prouff and Rivain [30] in several respects. First, they consider more general distributions to sample noisy leakage from, rather than just Gaussian [11] or Bernoulli leakage [18]. Moreover, they remove the limitation to one-bit observations, allowing the adversary to observe intermediate variables of any bitsize. Finally, they also extend the notion of leakage to take computation, rather than data, into account, following the *only*

computation leaks information principle introduced by Micali and Reyzin [24]. They also offer the first proof of security for a masked algorithm in this model, although it relies on leak-free components and a relatively weak adversary model.

In a different approach, Ishai, Sahai and Wagner [21] introduce the *t-threshold probing model*, in which the adversary receives the exact value of *at most t internal variables* (of his choice) in the computation. At the same time, they describe a transformation that turns any boolean circuit C secure in the *black-box model* into a circuit C' secure in the t-threshold probing model.

In practice, the noisy leakage model is often thought of as more realistic, since experimental physical leakage is noisy [23]. In particular, although the t-threshold probing model enables the adversary to observe exact values rather than noisy ones, it is not more powerful than the noisy leakage model, since the models also differ in the number of observations allowed to the adversary. The relationship between the two models was recently clarified by Duc, Dziembowski and Faust [14]. They advantageously recast the noisy leakage in the more classic statistical security model and show that security in the extended noisy leakage model of [30], fixed to capture chosen plaintext attacks, can be reduced to security in the t-threshold probing model of [21], in which security proofs are much more direct. In addition, the reduction does not rely on the existence of leak-free components. Thus, proving the security of a cryptosystem in the t-threshold probing model automatically ensures its security in the more realistic noisy leakage model.

In both models, only the values of intermediate variables are usually considered when determining the security order of an implementation. However, Balash et al. [2] show that this value-based leakage model does not fully capture some real-world scenarios in which additional physical leaks can occur, namely *glitches* or *transition-based* leakage, leaking information about more than one intermediate variable in a single observation. As a consequence, a perfectly masked algorithm secure in the value-based model can succumb to first-order attacks in these finer-grained leakage models.

Program Verification. Many tools aimed at proving the security of masked algorithms in the t-threshold probing model have recently appeared [9,15–17,25]. Some [9,25] use type systems to propagate sensitivity marks along the programs, but such approaches are not complete [16] and many programs are thus incorrectly typed as secure. Others take the underlying probability distributions into account, but can only handle low masking orders (typically orders 1 and 2), even on small programs.

Contributions. In this paper, we develop automated methods to prove the security of masked implementations in the t-threshold probing model, both for value-based and transition-based leakage. More specifically, our theoretical contributions are three-fold: i. We provide a formal characterization of security in the t-threshold probing model as a combination of variants of two well-known properties in programming languages: t-non-interference and functional

equivalence; ii. We provide algorithms that construct bijections between an adversary observation and a distribution that is trivially independent from the secret inputs, thereby proving that the adversary observation is independent from secret inputs; iii. We provide algorithms that make use of the constructed bijections to extend sets of observations with additional observations that do not give the adversary any more information about the secrets, thereby reducing greatly the number of non-interference proofs that need to be performed in order to prove a whole program t-non-interfering. As a practical contribution, we implement our algorithms and apply them to various masked implementations of AES, and a masked implementation of MAC-Keccak. Pleasingly, our tools are able to successfully analyze first-order masked implementations of AES (in a couple of minutes), 2 rounds of second-order masked implementations of AES at level 2 (in around 22 minutes), and masked implementations of multiplication, up to order 5 (in 45s). Our experiments allow us to rediscover several known attacks ([12,13]) on flawed implementations, to check that proposed fixes, when they exist, are indeed secure, and finally to discover new attacks on flawed implementations ([33]). We also discuss how our approach and tool can easily be adapted to deal with stronger leakage models capturing both transition-based leakage and leakage due to glitches, and illustrate it by studying the security of variants of secure field multiplication in the transition-based leakage model.

Putting our work in perspective, we deliberately focus on algorithmic methods that are able to cover large spaces of observation sets very efficiently, and without any assumption on the program. Although our results demonstrate that such methods can perform surprisingly well in practice, their inherent limitations with respect to scalability remain. A common strategy to address scalability issues is to develop compositional techniques. This could be done, for instance, in the context of a masking compiler, whose proof of security proceeds by showing that each gadget is secure, and that gadgets are combined securely. Assumptions on the structure of the masked algorithm could also be made that would allow such compositional reasoning. In this light, our algorithmic methods can be seen as focusing primarily on proving that core gadgets are secure with respect to a widely-used notion of security.

Outline. We first review previous uses of formal methods to prove similar properties (Section 2). In Sections 3 and 4, we describe our algorithms. In Section 5, we evaluate the practicality of our approach by implementing our algorithms in the framework provided by EasyCrypt [6], and testing the performance of our implementation on representative examples from the literature.

2 Language-Based Techniques for Threshold Security in the t-Threshold Probing Model

In this paper, we rephrase security in the t-threshold probing model by defining the notion of *t-non interference*, which is based on the notions of probabilistic non-interference used for verifying information-flow properties in language-based

security. We first define a general notion of program equivalence. Two probabilistic programs p_1 and p_2 are said to be $(\mathcal{I}, \mathcal{O})$-*equivalent*, denoted $p_1 \sim_{\mathcal{I}}^{\mathcal{O}} p_2$, whenever the probability distributions on $\mathcal{O}$ defined by p_1 and p_2, conditioned by the assumptions on input variables encoded in $\mathcal{I}$, are equal.

This notion of equivalence subsumes the two more specialized notions we consider here: *functional equivalence* and *t-non-interference*. Two programs p and $\bar{p}$ are said to be functionally equivalent when they are $(\mathcal{I}, \mathcal{Z})$-equivalent with $\mathcal{Z}$ all output variables, and $\mathcal{I}$ all input variables. A program $\bar{p}$ is said to be *t-non-interfering* with respect to a set of secret input variables $\mathcal{I}_{\mathsf{sec}}$ and a set of *observations* $\mathcal{O}$ when $\bar{p}(s_0, \cdot)$ and $\bar{p}(s_1, \cdot)$ are $(\mathcal{I}_{\mathsf{pub}}, \mathcal{O})$-equivalent (with $\mathcal{I}_{\mathsf{pub}} = \mathcal{I} \setminus \mathcal{I}_{\mathsf{sec}}$ the set of non-secret input variables) for any values s_0 and s_1 of the secret input variables.

We now give an indistinguishability-based definition of the t-threshold probing model. In this model, the challenger randomly chooses two secret values s_0 and s_1 (representing for instance two different values of the secret key) and a bit b according to which the leakage will be produced: the output computation always uses secret s_0, but the adversary observations are computed using s_b. The adversary $\mathcal{A}$ is allowed to query an oracle with chosen instances of public arguments, along with a set of at most t intermediate variables (adaptively or non-adaptively chosen); such queries reveal their output and the values of the intermediate variables requested by the adversary. We say that $\mathcal{A}$ wins if he guesses b.

We now state the central theorem to our approach, heavily inspired by Duc, Dziembowski and Faust [14] and Ishai, Sahai and Wagner [21].

Theorem 1. *Let p and $\overline{p}$ be two programs. If p and $\overline{p}$ are functionally equivalent and $\overline{p}$ is t-non-interfering, then for every adversary $\mathcal{A}$ against $\overline{p}$ in the t-threshold probing model, there exists an adversary $\mathcal{S}$ against p in the black-box model, such that*

$$\Delta(\mathcal{S} \overset{bb}{\leftrightarrows} p, \mathcal{A} \overset{thr}{\leftrightarrows} \overline{p}) = 0$$

where $\Delta(\cdot\,;\cdot)$ denotes the statistical distance[1]*.*

Proof. Since p and $\overline{p}$ are functionally equivalent, we have $\Delta(\mathcal{S} \overset{bb}{\leftrightarrows} p, \mathcal{S} \overset{bb}{\leftrightarrows} \overline{p}) = 0$ for all black-box adversary $\mathcal{S}$, and we only have to prove that there exists an $\mathcal{S}$ such that $\Delta(\mathcal{S} \overset{bb}{\leftrightarrows} \overline{p}, \mathcal{A} \overset{thr}{\leftrightarrows} \overline{p}) = 0$. We simply construct a simulator $\mathcal{S}'$ that simulates the leakage for $\mathcal{A}$, and build $\mathcal{S}$ by composing them. The simulator receives as inputs the public variables that are used for the execution of $\overline{p}$, and the output of $\overline{p}$, but not the t intermediate values corresponding to the observation set $\mathcal{O}$. Since $\overline{p}$ is t-non-interfering, the observations do not depend on the secret variables that are used for the execution of $\overline{p}$, and the simulator can choose arbitrary values for the secret variables, run $\overline{p}$ on these values and the public variables given as inputs, and output the requested observations. □

[1] The theorem can be lifted to the noisy leakage model using Corollary 1 from [14], using a small bound on the statistical distance instead.

Following Theorem 1, we propose algorithms to prove functional equivalence (details can be found in the long version of this document [3]) and t-non-interference properties of probabilistic programs, thereby reducing the security of masked implementations in the t-threshold probing model to the black-box security of the algorithms they implement.

In the following, we provide an overview of language-based techniques that could be used to verify the assumptions of Theorem 1, and to motivate the need for more efficient techniques. First, we introduce mild variants of two standard problems in programming languages, namely information-flow checking and equivalence checking, which formalize the assumptions of Theorem 1. Then, we present three prominent methods to address these problems: type systems (which are only applicable to information-flow checking), model counting, and relational logics. Finally, we discuss efficiency issues and justify the need for efficient techniques.

2.1 Problem Statement and Setting

The hypotheses of Theorem 1 can be seen as variants of two problems that have been widely studied in the programming language setting: equivalence checking and information-flow checking. Equivalence checking is a standard problem in program verification, although it is generally considered in the setting of deterministic programs, whereas we consider probabilistic programs here. Information-flow checking is a standard problem in language-based security, although it usually considers flows from secret inputs to public outputs, whereas we consider flows from secret inputs to intermediate values here.

Both problems can be construed as instances of relational verification. For clarity, we formalize this view in the simple case of straightline probabilistic programs. Such programs are sequences of random assignments and deterministic assignments, and have distinguished sets of input and output variables. Given a program p, we let $\mathsf{IVar}(p)$, $\mathsf{OVar}(p)$, and $\mathsf{PVar}(p)$ denote the sets of input, output, and intermediate variables of p. Without loss of generality, we assume that programs are written in single static assignment (SSA) form, and in particular, that program variables appear exactly once on the left hand side of an assignment, called their defining assignment—one can very easily transform an arbitrary straightline program into an equivalent straightline program in SSA form. Assuming that programs are in SSA form, we can partition $\mathsf{PVar}(p)$ into two sets $\mathsf{DVar}(p)$ and $\mathsf{RVar}(p)$ of deterministic and probabilistic variables, where a variable is probabilistic if it is defined by a probabilistic assignment, and is deterministic otherwise. Let $\mathcal{V}$ denote the set of program values (we ignore typing issues). Each program p can be interpreted as a function:

$$[\![p]\!] : \mathcal{D}(\mathcal{V}^{\kappa}) \to \mathcal{D}(\mathcal{V}^{\ell+\ell'})$$

where $\mathcal{D}(T)$ denotes the set of discrete distributions over a set T, and κ, ℓ and ℓ' respectively denote the sizes of $\mathsf{IVar}(p)$, $\mathsf{PVar}(p)$ and $\mathsf{OVar}(p)$. The function $[\![p]\!]$ takes as input a joint distribution on input variables and returns a joint

distribution on all program variables, and is defined inductively in the expected way. Furthermore, one can define for every subset $\mathcal{O}$ of $\mathsf{PVar}(p)$ of size m a function:

$$[\![p]\!]_{\mathcal{O}} : \mathcal{D}(\mathcal{V}^{\kappa}) \to \mathcal{D}(\mathcal{V}^{m})$$

that computes, for each $\boldsymbol{v} \in \mathcal{V}^{\kappa}$, the marginal distributions of $[\![p]\!](\boldsymbol{v})$ with respect to $\mathcal{O}$.

We can now define the information-flow checking problem formally: a program p is non-interfering with respect to a partial equivalence relation $\Phi \subseteq \mathcal{D}(\mathcal{V}^{\kappa}) \times \mathcal{D}(\mathcal{V}^{\kappa})$ (in the following, we write $\Phi\ \mu_1\ \mu_2$ to mean $(\mu_1, \mu_2) \in \Phi$), and a set $\mathcal{O} \subseteq \mathsf{PVar}(p)$, or $(\Phi, \mathcal{O})$-non-interfering, iff $[\![p]\!]_{\mathcal{O}}(\mu_1) = [\![p]\!]_{\mathcal{O}}(\mu_2)$ for every $\mu_1, \mu_2 \in \mathcal{D}(\mathcal{V}^{\kappa})$ such that $\Phi\ \mu_1\ \mu_2$. In this case, we write $\mathsf{NI}_{\Phi,\mathcal{O}}(p)$. Moreover, let $\mathbb{O}$ be a set of subsets of $\mathsf{PVar}(p)$, that is $\mathbb{O} \subseteq \mathcal{P}(\mathsf{PVar}(p))$; we say that p is $(\Phi, \mathbb{O})$-non-interfering, if it is $(\Phi, \mathcal{O})$-non-interfering for every $\mathcal{O} \in \mathbb{O}$.

Before relating non-interference with security in the t-threshold probing models, we briefly comment on the nature of Φ. In the standard, deterministic, setting for non-interference, variables are generally marked as secret or public—in the general case, they can be drawn from a lattice of security levels, but this is not required here. Moreover, Φ denotes low equivalence, where two tuples of values $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ are low-equivalent if they coincide on public variables. The notion of low-equivalence has a direct counterpart in the probabilistic setting: two distributions μ_1 and μ_2 are low equivalent iff their marginal distributions with respect to public variables are equal. However, non-interference of masked implementations is often conditioned by well-formedness conditions on inputs; for instance, the inputs must consist of uniformly distributed, t-wise independent values. In this case, Φ is defined in such a way that two distributions are related by Φ iff they are well-formed and low equivalent.

There is a direct interpretation of t-threshold probing security as a non-interference property. We say that a program p is (Φ, t)-non-interfering if it is $(\Phi, \mathcal{O})$-non-interfering for all subsets $\mathcal{O}$ of $\mathsf{PVar}(p)$ with size smaller than t (we write $\mathcal{O} \in \mathcal{P}_{\leq t}(\mathsf{PVar}(p))$ in the following). Then a program p is secure in the t-threshold probing model (with respect to a relation Φ) iff it is (Φ, t)-non-interfering.

In order to capture t-threshold probing security in the transition-based leakage model, we rely on a partial function next that maps program variables to their successors. For programs that have been translated into SSA form, all program variables are of the form x_i, where x is a variable of the original program, and i is an index—typically a program line number. The successor of such a variable x_i, when it exists, is a variable of the form x_j where j is the smallest index such that $i < j$ and x_j is a program variable. Then, we say that a program p is (Φ, t)-non-interfering in the transition-based model, written $\mathsf{NI}_{\Phi,t,\mathsf{succ}}(p)$, iff p is $(\Phi, \mathcal{O} \cup \mathsf{next}(\mathcal{O}))$-non-interfering for every subset of $\mathsf{PVar}(p)$ with size smaller than t. Then a program p is secure in the transition-based t-threshold

probing model (with respect to a relation Φ) iff it is (Φ, t)-non-interfering in the transition-based model.[2]

We now turn to program equivalence. For the sake of simplicity, we consider two programs p_1 and p_2 that have the same sets of input and output variables; we let $\mathcal{W}$ denote the latter. We let $[\![p]\!]_{\mathcal{W}}$ denote the function that computes for every initial distribution μ the marginal distribution of $[\![p]\!](\mu)$ with respect to $\mathcal{W}$. We say that p_1 and p_2 are equivalent with respect to a partial equivalence relation $\Phi \subseteq \mathcal{D}(\mathcal{V}^\kappa) \times \mathcal{D}(\mathcal{V}^\kappa)$, written $p_1 \sim_\Phi p_2$, iff $[\![p_1]\!]_{\mathcal{W}}(\mu) = [\![p_2]\!]_{\mathcal{W}}(\mu)$ for every distribution μ such that $\Phi\ \mu\ \mu$.

For the sake of completeness, we point out that both notions are subsumed by the notion of $(\Phi, \mathcal{O})$-equivalence. Specifically, we say that programs p_1 and p_2 are $(\Phi, \mathcal{O})$-equivalent, written $p_1 \sim^{\mathcal{O}}_{\Phi} p_2$, iff $[\![p_1]\!]_{\mathcal{O}}(\mu_1) = [\![p_2]\!]_{\mathcal{O}}(\mu_2)$ for every two distributions μ_1 and μ_2 such that $\Phi\ \mu_1\ \mu_2$. Therefore, both equivalence checking and information-flow checking can be implemented using as subroutine any sound algorithm for verifying that $p_1 \sim^{\mathcal{O}}_{\Phi} p_2$.

2.2 Type-Based Approaches

Information-flow type systems are a class of type systems that enforce non-interference by tracking dependencies between program variables and rejecting programs containing illicit flows. There are multiple notions of non-interference (termination-sensitive, termination-insensitive, or bisimulation-based) and forms of information-flow type systems (for instance, flow-sensitive, or flow-insensitive); we refer the reader to [32] for a survey. For the purpose of this paper, it is sufficient to know that information-flow type systems for deterministic programs assign to all program variables a level drawn from a lattice of security levels which includes a level of public variables and secret variables. In the same vein, one can develop information-flow type systems to enforce probabilistic non-interference; broadly speaking, such type systems distinguish between public values, secret values, and uniformly distributed values. Following these ideas, Moss et al. [25] pioneer the application of information-flow type systems to masking. They use the type system as a central part in a masking compiler that transforms an input program into a functionally equivalent program that is resistant to first-order DPA. Their technique can readily be extended to prove non-interference with respect to a single observation set.

Because they are implemented with well-understood tools (such as data flow analyses) and are able to handle large programs extremely fast, information-flow type systems provide an appealing solution that one would like to use for higher-order DPA. However, the semantic information carried by types is inherently attached to individual values, rather than tuples of values, and there is

[2] Similarly, glitches could be captured by considering that each observation leaks four values: the values of the arguments, and the old and new values of the wire or register. More fine-grained leakage models depending on implementation details and combining value-based, transition-based and glitch-based leakage could also be considered.

no immediately obvious way to devise an information-flow type system even for second-order DPA. Notwithstanding, it is relatively easy to devise a sound method for verifying resistance to higher-order DPA using an information-flow type system in the style of [25]. The basic idea is to instrument the code of the original program with assignments $w := x_1 \parallel \ldots \parallel x_t$, where w is a fresh program variable, $x_1 \ldots x_t$ are variables of the original program, and t is the order for which resistance is sought; we let p' denote the instrumented program. Clearly, a program p is secure at order t iff for every initial values $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$, $[\![p']\!]_{\{w\}}(\boldsymbol{v_1}) = [\![p']\!]_{\{w\}}(\boldsymbol{v_2})$ where w ranges over the set of fresh variables that have been introduced by the transformation. It is then possible to use an information-flow type system in the spirit of [25] to verify that c' satisfies non-interference with respect to output set $\{w\}$. However, this transformational approach suffers from two shortcomings: first, a more elaborate type system is required for handling concatenation with sufficient accuracy; second, and more critically, the transformation induces an exponential blow-up in the size of programs.

In a slightly different context, Pettai and Laud [29] use a type-system to prove non-interference of a limited number of adversary observations imposed by their adversary model in the multi-party computation scenario. They do so by propagating information regarding linear dependencies on random variables throughout their arithmetic circuits and progressively replacing subcircuits with random gates. Because of the limited number of possible adversary observations their model imposes, they do not run into the same scalability issues we deal with in this paper. However, their techniques for dealing with active adversaries may be useful for verifying masking-based countermeasures in the presence of fault injection attacks.

2.3 SMT-Based Methods

There have been a number of works that use SMT solvers to achieve more flexible analysis of masked implementations.

Bayrak et al. [9] develop an SMT-based method for analyzing the sensitivity of sequences of operations. Informally, the notion of sensitivity characterizes whether a variable used to store an intermediate computation in the sequence of operations depends on a secret and is statistically independent from random variables. Their approach is specialized to first-order masking, and suffers from some scalability issue—in particular, they report analysis of a single round of AES.

Eldib, Wang and Schaumont develop an alternative tool, SCSniffer [16], that is able to analyze masked implementations at orders 1 and 2. Their approach is based on model counting [20]: to prove that a set of probabilistic expressions is distributed independently from a set of secrets, model-counting-based tools count the number of valuations of the secrets that yield each possible value of the observed expressions and checks that that number is indeed independent from the secret. This process in itself is inherently exponential in the size of the observed expressions, even when only one such observation is considered. To overcome this issue, SCSniffer implements an incremental approach for reducing

the size of such expressions when they contain randomness that is syntactically independent from the rest of the program. This incremental approach is essential to analyzing some of their examples, but it is still insufficient for analyzing complete implementations: for instance, SCSniffer can only analyze one round of (MAC-)Keccak whereas our approach is able to analyze the full 24 rounds of the permutation. The additional power of our tool is derived from our novel technique: instead of explicitly counting solutions to large boolean systems, our tool simply constructs a bijection between two distributions, one of which is syntactically independent from the secrets. Although the complexity of this process still depends on the size of expressions (and in particular in the number of randomly sampled variables they contain), it is only polynomial in it, rather than exponential. In addition, the approach, as it is used in Sleuth and SCSniffer, is limited to the study of boolean programs or circuits, where all variables are 1 bit in size. This leads to unwieldy program descriptions and artificially increases the size of expressions, thereby also artificially increasing the complexity of the problem. Our approach bypasses this issue by considering abstract algebraic expressions rather than specific types. This is only possible because we forego explicit solution counting. Moreover, SCSniffer requires to run the tool at all orders $d \leq t$ to obtain security at level t. In contrast, we achieve the same guarantees in a single run. This is due to the fact that the exclusive-or of observed variables is used for model counting rather than their joint distribution. Our approach yields proofs of t-non-interference directly by considering the joint distribution of observed variables. Finally, we contribute a technique that helps reduce the practical complexity of the problem by extending proofs of independence for a given observation set into a proof of independence for many observation sets at once. This process is made less costly by the fact that we can efficiently check whether a proof of independence is still valid for an extended observation set, but we believe it would apply to techniques based on model-counting given the same ability.

All of these differences lead to our techniques greatly outperforming existing approaches when it comes to practical examples. For example, even considering only masking at order 1, where it takes SCSniffer 10 minutes to prove a masked implementation of one round of Keccak (implemented bit-by-bit), it takes our tool around 7 minutes to prove the full 24 rounds of the permutation (implemented on 64-bit words as in reference implementations), and around 2 minutes to verify a full implementation of AES (including its key schedule).

2.4 Relational Verification

A more elaborate approach is to use program verification for proving non-interference and equivalence of programs. Because these properties are inherently relational—that is, they either consider two programs or two executions of the same program—the natural verification framework to establish such properties is relational program logic. Motivated by applications to cryptography, Barthe, Grégoire and Zanella-Bèguelin [8] introduce pRHL, a probabilistic Relational Hoare Logic that is specifically tailored for the class of probabilistic programs

considered in this paper. Using pRHL, $(\phi, \mathcal{O})$-non-interference a program p is captured by the pRHL judgment:

$$\{\phi\}p \sim p\{\bigwedge_{y \in \mathcal{O}} y\langle 1\rangle = y\langle 2\rangle\}$$

which informally states that the joint distributions of the variables $y \in \mathcal{O}$ coincide on any two executions (which is captured by the logical formula $y\langle 1\rangle = y\langle 2\rangle$) that start from initial memories related by Φ.

Barthe et al. [7] propose an automated method to verify the validity of such judgments. For clarity of our exposition, we consider the case where p is a straightline code program. The approach proceeds in three steps:

1. transform the program p into a semantically equivalent program which performs a sequence of random assignments, and then a sequence of deterministic assignments. The program transformation repeatedly applies eager sampling to pull all probabilistic assignments upfront. At this stage, the judgement is of the form

$$\{\phi\}S; D \sim S; D\{\bigwedge_{y \in \mathcal{O}} y\langle 1\rangle = y\langle 2\rangle\}$$

   where S is a sequence of probabilistic assignments, and D is a sequence of deterministic assignments;
2. apply a relational weakest precondition calculus to D the deterministic sequence of assignments; at this point, the judgment is of the form

$$\{\phi\}S \sim S\{\bigwedge_{y \in \mathcal{O}} e_y\langle 1\rangle = e_y\langle 2\rangle\}$$

   where e_y is an expression that depends only on the variables sampled in S and on the program inputs;
3. repeatedly apply the rule for random sampling to generate a verification condition that can be discharged by SMT solvers. Informally, the rule for random sampling requires finding a bijection between the domains of the distribution from which values are drawn, and proving that a formula derived from the post-condition is valid. We refer to [8] and [7] for a detailed explanation of the rule for random sampling. For our purposes, it is sufficient to consider a specialized logic for reasoning about the validity of judgments of the form above. We describe such a logic in Section 3.1.

Note that there is a mismatch between the definition of (Φ, t)-non-interference used to model security in the t-threshold probing model, and the notion of $(\phi, \mathcal{O})$-non-interference modelled by pRHL. In the former, Φ is a relation over distributions of memories, whereas in the latter ϕ is a relation over memories. There are two possible approaches to address this problem: the first is to develop a variant of pRHL that supports a richer language of assertions; while possible, the

resulting logic might not be amenable to automation. A more pragmatic solution, which we adopt in our tool, is to transform the program p into a program $i; p$, where i is some initialization step, such that p is $(\Phi, \mathcal{O})$ non-interfering iff $i; p$ is $(\phi, \mathcal{O})$ non-interfering for some pre-condition ϕ derived from Φ.

In particular, i includes code marked as non-observable that *preshares* any input or state marked as secret,[3] and fully observable code that simply shares public inputs. The code for sharing and presharing, as well as a simple example of this transformation are given in Appendix A.

3 A Logic for Probabilistic Non-Interference

In this section, we propose new verification-based techniques to prove probabilistic non-interference statements. We first introduce a specialized logic to prove a vector of probabilistic expressions independent from some secret variables. We then explain how this logic specializes the general approach described in Section 2.4 to a particular interesting case. Finally, we describe simple algorithms that soundly construct derivations in our logic.

3.1 Our Logic

Our logic shares many similarities with the equational logic developed in [5] to reason about equality of distributions. In particular, it considers equational theories over multi-sorted signatures.

A multi-sorted signature is defined by a set of types and a set of operators. Each operator has a signature $\sigma_1 \times \ldots \times \sigma_n \to \tau$, which determines the type of its arguments, and the type of the result. We assume that some operators are declared as invertible with respect to one or several of their arguments; informally, a k-ary operator f is invertible with respect to its i-th argument, or i-invertible for short, if, for any $(x_j)_{i \neq j}$ the function $f(x_0, \ldots, x_{i-1}, \cdot, x_{i+1}, \ldots, x_k)$ is a bijection. If f is i-invertible, we say that its i-th argument is an *invertible argument of* f.

Expressions are built inductively from two sets $\mathcal{R}$ and $\mathcal{X}$ of probabilistic and deterministic variables respectively, and from operators. Expressions are (strongly) typed. The set of deterministic (resp. probabilistic) variables of a vector of expressions $\boldsymbol{e}$ is denoted as $\mathsf{dvar}(\boldsymbol{e})$ (resp. $\mathsf{rvar}(\boldsymbol{e})$). We say that an expression e is *invertible in* x whenever $\forall i\ j,\ x \notin \mathsf{rvar}(e_i^j)$, we have $e = f_1(\ldots, e^1_{i_1-1}, f_2(\ldots f_n(\ldots, e^n_{i_n-1}, x, \ldots)\ldots), \ldots)$, and each f_j is i_j-invertible.

We equip expressions with an equational theory $\mathcal{E}$. An equational theory is a set of equations, where an equation is a pair of expressions of the same type. Two expressions e and e' are provably equal with respect to an equational theory $\mathcal{E}$, written $e \doteq_{\mathcal{E}} e'$, if the equation $e \doteq_{\mathcal{E}} e'$ can be derived from the standard rules of multi-sorted equational logic: reflexivity, symmetry, transitivity, congruence, and instantiation of axioms in $\mathcal{E}$. Such axioms can be used, for example, to equip types with particular algebraic structures.

[3] This corresponds to Ishai, Sahai and Wagner's *input encoders* [21].

Expressions have a probabilistic semantics. A valuation ρ is a function that maps deterministic variables to values in the interpretation of their respective types. The interpretation $[\![e]\!]_\rho$ of an expression is a discrete distribution over the type of e; informally, $[\![e]\!]_\rho$ samples all random variables in e, and returns the usual interpretation of e under an extended valuation ρ, ρ' where ρ' maps each probabilistic variable to a value of its type. The definition of interpretation is extended to tuples of expressions in the obvious way. Note that, contrary to the deterministic setting, the distribution $[\![(e_1, \ldots, e_k)]\!]_\rho$ differs from the product distribution $[\![e_1]\!]_\rho \times \ldots \times [\![e_k]\!]_\rho$. We assume that the equational theory is consistent with respect to the interpretation of expressions.

Judgments in our logic are of the form $(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e}$, where $\boldsymbol{e}$ is a set of expressions and $(\boldsymbol{x_L}, \boldsymbol{x_H})$ partitions the deterministic variables of $\boldsymbol{e}$ into public and private inputs, that is, $\mathsf{dvar}(\boldsymbol{e}) \subseteq \boldsymbol{x_L} \uplus \boldsymbol{x_H}$. A judgment $(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e}$ is valid iff the identity of distributions $[\![\boldsymbol{e}]\!]_{\rho_1} = [\![\boldsymbol{e}]\!]_{\rho_2}$ holds for all valuations ρ_1 and ρ_2 such that $\rho_1(x) = \rho_2(x)$ for all $x \in \boldsymbol{x_L}$.

A proof system for deriving valid judgments is given in Figure 1. Rule (Indep) states that a judgment is valid whenever all the deterministic variables in expressions are public. Rule (Conv) states that one can replace expressions by other expressions that are provably equivalent with respect to the equational theory $\mathcal{E}$. Rule (Opt) states that, whenever the only occurrences of a random variable r in $\boldsymbol{e}$ are as the i-th argument of some fixed application of an i-invertible operator f where f's other arguments are some $(e_j)_{i \neq j}$, then it is sufficient to derive the validity of the judgment where r is substituted for $f(e_0, \ldots, e_{i-1}, r, e_{i+1}, \ldots, e_k)$ in $\boldsymbol{e}$. The soundness of rule (Opt) becomes clear by remarking that the distributions $[\![f(e_0, \ldots, e_{i-1}, r, e_{i+1}, \ldots, e_k)]\!]$ and $[\![r]\!]$ are equal, since f is i-invertible and r is uniform random and does not appear in any of the e_j. Although the proof system can be extended with further rules (see, for example [5]), these three rules are in fact sufficient for our purposes.

$$\frac{\mathsf{dvar}(\boldsymbol{e}) \cap \boldsymbol{x_H} = \emptyset}{(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e}}\ (\textsc{Indep}) \qquad \frac{(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e'} \quad \boldsymbol{e} \doteq_{\mathcal{E}} \boldsymbol{e'}}{(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e}}\ (\textsc{Conv})$$

$$\frac{(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e} \quad f \text{ is } i\text{-invertible} \quad r \in \mathcal{R} \quad r \notin \mathsf{rvar}(e_0, \ldots, e_{i-1}, e_{i+1}, \ldots, e_k)}{(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash \boldsymbol{e}[f(e_0, \ldots, e_{i-1}, r, e_{i+1}, \ldots, e_k)/r]}\ (\textsc{Opt})$$

Fig. 1. Proof system for non-interference

3.2 From Logical Derivations to Relational Judgments

In Section 2.4, we have shown that the problem of proving that a program is $(\Phi, \mathcal{O})$-non-interfering could be reduced to proving relational judgements of the form $\{\phi\} S \sim S\{\bigwedge_{y \in \mathcal{O}} e_y\langle 1 \rangle = e_y\langle 2 \rangle\}$ where S is a sequence of random samplings, e_y is an expression that depends only on the variables sampled in S and on the program inputs, and ϕ is a precondition derived from Φ after

the initial sharing and presharing code is inserted, and exactly captures low-equivalence on the program's inputs. We now show that proving such judgments can in fact be reduced to constructing a derivation in the logic from Section 3.1. Indeed, since both sides of the equalities in the postcondition are equal, it is in fact sufficient to prove that the $(e_y)_{y \in \mathcal{O}}$ are independent from secret inputs: since public inputs are known to be equal and both programs are identical, the postcondition then becomes trivially true. In particular, to prove the judgment $\{\bigwedge_{x \in \boldsymbol{x_L}} x\langle 1\rangle = x\langle 2\rangle\} S \sim S \{\bigwedge_{y \in \mathcal{O}} e_y\langle 1\rangle = e_y\langle 2\rangle\}$, it is in fact sufficient to find a derivation of $(\boldsymbol{x_L}, \boldsymbol{x_H}) \vdash (e_y)_{y \in \mathcal{O}}$, where $\boldsymbol{x_H}$ is the complement of $\boldsymbol{x_L}$ in the set of all program inputs. An example detailing this reasoning step is discussed in Appendix A.

3.3 Our Algorithms

We now describe two algorithms that soundly derive judgments in the logic. Throughout this paper, we make use of unspecified choose algorithms that, given a set X, return an $x \in X$ or $\bot$ if $X = \emptyset$. We discuss our chosen instantiations where valuable.

Our simplest algorithm (Algorithm 1) works using only rules (Indep) and (Opt) of the logic. Until (Indep) applies, Algorithm 1 tries to apply (Opt), that is, to find $(\boldsymbol{e'}, e, r)$ such that $r \in \mathcal{R}$ and e is invertible in r and $\boldsymbol{e} = \boldsymbol{e'}[e/r]$; if it succeeds, it then performs a recursive call on e' else it fails. Remark that the conditions are sufficient to derive the validity of e from the validity of e' using successive applications of the (Opt) rule.

The result of the function $(\boldsymbol{h})$ can be understood as a compact representation of the logical derivation. Such compact representations of derivations become especially useful in Section 4, where we efficiently extend sets of observed expressions, but can also be used, independently of performance, to construct formal proof trees if desired.

Algorithm 1. Proving Probabilistic Non-Interference: A Simple Algorithm

```
1: function NI_{R,x_H}(e)                ▷ the joint distribution of e is independent from x_H
2:     if ∀x ∈ dvar(e). x ∉ x_H then
3:         return Indep
4:     (e', e, r) ← choose({(e', e, r) | e is invertible in r ∧ r ∈ R ∧ e = e'[e/r]})
5:     if (e', e, r) ≠ ⊥ then
6:         return Opt(e, r) : NI_{R,x_H}(e')
7:     return ⊥
```

This algorithm is sound, since it returns a derivation $\boldsymbol{h}$ constructed after checking each rule's side-conditions. However, it is incomplete and may fail to construct valid derivations. In particular, it does not make use of rule (Conv).

Our second algorithm (Algorithm 2) is a slight improvement on Algorithm 1 that makes restricted use of the (Conv) rule: when we cannot find a suitable $(\boldsymbol{e'}, e, r)$, we normalize algebraic expressions as described in [1], simplifying

expressions and perhaps revealing potential applications of the (Opt) rule. We use only algebraic normalization to avoid the need for user-provided hints, and even then, only use this restricted version of the (Conv) rule as a last resort. This is for two reasons: first, ring normalization may prevent the use of some $(\boldsymbol{e'}, e, r)$ triples in later recursive calls (for example, the expression $(a + r) \cdot r'$ gets normalized as $a \cdot r' + r \cdot r'$, which prevents the substitution of $a + r$ by r); second, the normalization can be costly and negatively impact performance.

Algorithm 2. Proving Probabilistic Non-Interference: A More Precise Algorithm

```
1: function NI_{R,x_H}(e, b)        ▷ the joint distribution of e is independent from x_H
2:     if ∀x ∈ dvar(e). x ∉ x_H then
3:         return Indep
4:     (e', e, r) ← choose({(e', e, r) | e is invertible in r ∧ r ∈ R ∧ e = e'[e/r]})
5:     if (e', e, r) ≠ ⊥ then
6:         return Opt(e, r) : NI_{R,x_H}(e', b)
7:     else if b then
8:         e ← ring_simplify(e)
9:         return Conv : NI_{R,x_H}(e, false)
10:    return ⊥
```

In practice, we have found only one example where Algorithm 1 yields false negatives, and we have not found any where Algorithm 2 fails to prove the security of a secure implementation. In the following, we use $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(X)$ the function from Algorithm 2 with b initially true. In particular, the implementation described and evaluated in Section 5 relies on this algorithm.[4]

Discussion. We observe that Algorithm 2 can only be refined in this way because it works directly on program expressions. In particular, any abstraction, be it type-based or otherwise, could prevent the equational theory from being used to simplify observed expressions. Further refinements are theoretically possible (in particular, we could also consider a complete proof system for the logic in Section 3.1), although they may be too costly to make use of in practice.

4 Divide-and-Conquer Algorithms Based on Large Sets

Even with efficient algorithms to prove that a program p is $(\mathcal{R}, \mathcal{O})$-non-interfering for some observation set $\mathcal{O}$, proving that p is t-non-interfering remains a complex task: indeed this involves proving $\mathsf{NI}_{\mathcal{R},\mathcal{O}}(p)$ for all $\mathcal{O} \in \mathcal{P}_{\leq t}(\mathsf{PVar}(\mathsf{p}))$. Simply

[4] Some of the longer-running experiments reported in Section 5 do make use of Algorithm 1 since their running time makes it impractical to run them repeatedly after algorithmic changes. However, Algorithm 2 only makes a difference when false positives occur, which is not the case on our long-running tests.

enumerating all possible observation sets quickly becomes intractable as p and t grow. Our main idea to solve this problem is based on the following fact: if $\mathsf{NI}_{\mathcal{R},\mathcal{O}}(p)$ then for every $\mathcal{O}' \subseteq \mathcal{O}$ we have $\mathsf{NI}_{\mathcal{R},\mathcal{O}'}(p)$. Therefore checking $\mathsf{NI}_{\mathcal{R},\mathcal{O}_i}(p)$ for every i can be done in a single step by checking $\mathsf{NI}_{\mathcal{R},\bigcup_i \mathcal{O}_i}(p)$.

Our goal is therefore to find fewer, larger observation sets $\mathcal{O}_1, \ldots, \mathcal{O}_k$ such that $\mathsf{NI}_{\mathcal{R},\mathcal{O}_k}(p)$ for all k and, for all $\mathcal{O} \in \mathcal{P}_{\leq t}(\mathsf{PVar(p)})$, $\mathcal{O}$ is a subset of at least one of the $\mathcal{O}_i$. Since this last condition is the contraposite of the Hitting Set problem [19], which is known to be NP-hard, we do not expect to find a generally efficient solution, and focus on proposing algorithms that prove efficient in practice.

We describe and implement several algorithms based on the observation that the sequences of derivations constructed to prove the independence judgments in Section 2 can be used to efficiently extend the observation sets with additional observations whose joint distributions with the existing ones is still independent from the secrets. We first present algorithms that perform such extensions, and others that make use of observation sets extended in this way to find a family $\mathcal{O}_1, \ldots, \mathcal{O}_k$ of observation sets that fulfill the condition above with k as small as possible.

4.1 Extending Safe Observation Sets

The $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}$ algorithm from Section 2 (Algorithm 2) allows us to identify sets X of expressions whose joint distribution is independent from variables in $\boldsymbol{x_H}$. We now want to extend such an X into a set X' that may contain more observable expressions and such that the joint distribution of X' is still independent from variables in $\boldsymbol{x_H}$.

First we define Algorithm 3, which rechecks that a derivation applies to a given set of expressions using the compact representation of derivations returned by algorithms 1 and 2: The algorithm simply checks that the consecutive rules encoded by $\boldsymbol{h}$ can be applied on $\boldsymbol{e}$. A key observation is that if $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}) = \boldsymbol{h}$ then $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e},\boldsymbol{h})$. Furthermore, if $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e},\boldsymbol{h})$ and $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}',\boldsymbol{h})$ then $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e} \cup \boldsymbol{e}',\boldsymbol{h})$.

Algorithm 3. Rechecking a derivation

```
function recheck_{R,x_H}(e, h)        ▷ Check that the derivation represented by h can be applied to e
    if h = INDEP then
        return ∀x ∈ dvar(e). x ∉ x_H
    if h = OPT(e, r) : h' then
        (e') ← choose({e' | e = e'[e/r]})
        if e' ≠ bot then
            return recheck_{R,x_H}(e', h')
    if h = CONV : h' then
        e ← ring_simplify(e)
        return recheck_{R,x_H}(e, h')
```

Algorithm 4. Extending the Observation using a Fixed Derivation

```
function extend_{R,x_H}(x, e, h)        ▷ find x' such that x ⊆ x' ⊆ e and h(x') is
                                          syntactically independent from x_H
    e ← choose(e)
    if recheck_{R,x_H}(e, h) then
        return extend_{R,x_H}((x, e), e \ {e}, h)
    else
        return extend_{R,x_H}(x, e \ {e}, h)
```

Secondly, we consider (as Algorithm 4) an extension operation that only adds expressions on which $\boldsymbol{h}$ can safely be applied as it is.

We also considered an algorithm that extends a set $\boldsymbol{x}$ with elements in $\boldsymbol{e}$ following $\boldsymbol{h}$ whilst also extending the derivation itself when needed. However, this algorithm induces a loss of performance due to the low proportion of program variables that can in fact be used to extend the observation set, wasting a lot of effort on attempting to extend the derivation when it was not in fact possible. Coming up with a good choose algorithm that prioritizes variables that are likely to be successfully added to the observation set, and with conservative and efficient tests to avoid attempting to extend the derivation for variables that are clearly not independent from the secrets are interesting challenges that would refine this algorithm, and thus improve the performance of the space splitting algorithms we discuss next.

In the following, we use $\mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{x}, \boldsymbol{e}, \boldsymbol{h})$ to denote the function from Algorithm 4, which is used to obtain all experimental results reported in Section 5.

4.2 Splitting the Space of Adversary Observations

Equipped with an efficient observation set extension algorithm, we can now attempt to accelerate the coverage of all possible sets of adversary observations to prove t-non-interference. The general idea of these coverage algorithms is to choose a set X of t observations and prove that the program is non-interfering with respect to X, then use the resulting derivation witness to efficiently extend X into an $\widehat{X}$ that contains (hopefully many) more variables. This $\widehat{X}$, with respect to which the program is known to be non-interfering, can then be used to split the search space recursively. In this paper, we consider two splitting strategies to accelerate the enumeration: the first (Algorithm 5) simply splits the observation space into $\widehat{X}$ and its complement before covering observations that straddle the two sets. The second (Algorithm 6) splits the space many-ways, considering all possible combinations of the sub-spaces when merging the sets resulting from recursive calls.

Pairwise Space-Splitting. Our first algorithm (Algorithm 5) uses its initial tuple X to split the space into two disjoint sets of observations, recursively descending into the one that does not supersede X and calling itself recursively to merge the two sets once they are processed separately.

Algorithm 5. Pairwise Space-Splitting

1: **function** $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{x}, d, \boldsymbol{e})$ ▷ every $\boldsymbol{x}, \boldsymbol{y}$ with $\boldsymbol{y} \in \mathcal{P}_{\leq d}(\boldsymbol{e})$ is independent of $\boldsymbol{x_H}$
2: **if** $d \leq |E|$ **then**
3: $\boldsymbol{y} \leftarrow \mathsf{choose}(\mathcal{P}_{\leq d}(\boldsymbol{e}))$
4: $\boldsymbol{h_{x,y}} \leftarrow \mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}((\boldsymbol{x}, \boldsymbol{y}))$ ▷ if $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}$ fails, raise error $\mathsf{CannotProve}\ (\boldsymbol{x}, \boldsymbol{y})$
5: $\widehat{\boldsymbol{y}} \leftarrow \mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{y}, \boldsymbol{e} \setminus \boldsymbol{y}, \boldsymbol{h_{x,y}})$ ▷ if $\boldsymbol{h_{x,y}} = \top$, use $\widehat{\boldsymbol{y}} = \boldsymbol{y}$
6: $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{x}, d, \boldsymbol{e} \setminus \widehat{\boldsymbol{y}})$
7: **for** $0 < i < d$ **do**
8: **for** $\boldsymbol{u} \in \mathcal{P}_{\leq i}(\widehat{\boldsymbol{y}})$ **do**
9: $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}((\boldsymbol{x}, \boldsymbol{u}), d - i, \boldsymbol{e} \setminus \widehat{\boldsymbol{y}})$

Theorem 2 (Soundness of Pairwise Space-Splitting). *Given a set $\mathcal{R}$ of random variables, a set $\boldsymbol{x_H}$ of secret variables, a set of expressions $\boldsymbol{e}$ and an integer $t > 0$, if $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}(\emptyset, t, \boldsymbol{e})$ succeeds then every $\boldsymbol{x} \in \mathcal{P}_{\leq t}(\boldsymbol{e})$ is independent from $\boldsymbol{x_H}$.*

Proof. The proof is by generalizing on $\boldsymbol{x}$ and d and by strong induction on $\boldsymbol{e}$. If $|\boldsymbol{e}| < d$, the theorem is vacuously true, and this base case is eventually reached since $\widehat{\boldsymbol{y}}$ contains at least d elements. Otherwise, by induction hypothesis, the algorithm is sound for every $\boldsymbol{e'} \subsetneq \boldsymbol{e}$. After line 5, we know that all t-tuples of variables in $\widehat{\boldsymbol{y}}$ are independent, jointly with $\boldsymbol{x}$, from the secrets. By the induction hypothesis, after line 6, we know that all t-tuples of variables in $\boldsymbol{e} \setminus \widehat{\boldsymbol{y}}$ are independent, jointly with $\boldsymbol{x}$, from the secrets. It remains to prove the property for t-tuples that have some elements in $\widehat{\boldsymbol{y}}$ and some elements in $\boldsymbol{e} \setminus \widehat{\boldsymbol{y}}$. The nested for loops at lines 7-9 guarantee it using the induction hypothesis. □

Worklist-Based Space-Splitting. Our second algorithm (Algorithm 6) splits the space much more finely given an extended safe observation set. The algorithm works with a worklist of pairs $(d, \boldsymbol{e})$ (initially called with a single element $(t, \mathcal{P}_{\leq t}(\mathsf{PVar}(p)))$). Unless otherwise specified, we lift algorithms seen so far to work with vectors or sets of arguments by applying them element by element. Note in particular, that the for loop at line 7 iterates over all vectors of n integers such that each element i_j is strictly between 0 and d_j.

Algorithm 6. Worklist-Based Space-Splitting

1: **function** $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}((d_j, \boldsymbol{e}_j)_{0 \leq j < n})$ ▷ every $\boldsymbol{x} = \bigcup_{0 \leq j < n} \boldsymbol{x}_j$ with $\boldsymbol{x}_j \in \mathcal{P}_{\leq d_j}(\boldsymbol{e}_j)$ is independent from $\boldsymbol{x_H}$
2: **if** $\forall j,\ d_j \leq |\boldsymbol{e}_j|$ **then**
3: $\boldsymbol{y}_j \leftarrow \mathsf{choose}(\mathcal{P}_{\leq d_j}(\boldsymbol{e}_j))$
4: $\boldsymbol{h} \leftarrow \mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\bigcup_{0 \leq j < n} \boldsymbol{y}_j)$ ▷ if $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}$ fails, raise error $\mathsf{CannotProve}\ (\bigcup \boldsymbol{y}_j)$
5: $\widehat{\boldsymbol{y}_j} \leftarrow \mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{y}_j, \boldsymbol{e}_j \setminus \boldsymbol{y}_j, \boldsymbol{h})$
6: $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}((d_j, \boldsymbol{e}_j \setminus \widehat{\boldsymbol{y}_j})_{0 \leq j < n})$
7: **for** j; $0 < i_j < d_j$ **do**
8: $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}(i_j, (\widehat{\boldsymbol{y}_j}, d_j - i_j, \boldsymbol{e}_j \setminus \widehat{\boldsymbol{y}_j}))$

Theorem 3 (Soundness of Worklist-Based Space-Splitting). *Given a set $\mathcal{R}$ of random variables, a set $\boldsymbol{x_H}$ of secret variables, a set of expressions $\boldsymbol{e}$ and an integer $t > 0$, if* $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}((t,\boldsymbol{e}))$ *succeeds then every $\boldsymbol{x} \in \mathcal{P}_{\leq t}(\boldsymbol{e})$ is independent from $\boldsymbol{x_H}$.*

Proof. As in the proof of Theorem 2, we start by generalizing, and we prove that, for all vector $(d_j, \boldsymbol{e}_j)$ with $0 < d_j$ for all j, if $\mathsf{check}_{\mathcal{R},\boldsymbol{x_H}}((d_j,\boldsymbol{e}_j))$ succeeds, then every $\boldsymbol{x} = \bigcup_{0\leq j<n} \boldsymbol{x}_j$ with $\boldsymbol{x}_j \in \mathcal{P}_{\leq d_j}(\boldsymbol{e}_j)$ is independent from $\boldsymbol{x_H}$. The proof is again by strong induction on the vectors, using an element-wise lexicographic order (using size order on the $\boldsymbol{e}$) and lifting it to multisets as a bag order. If there exists an index i for which $|\boldsymbol{e}_i| < d_i$, the theorem is vacuously true. Otherwise, we unroll the algorithm in a manner similar to that in Theorem 2. After line 5, we know that, for every j, every $\boldsymbol{x} \in \mathcal{P}_{\leq d_j}(\widehat{\boldsymbol{y}_j})$ is independent from x_H. After line 6, by induction hypothesis (for all j, $\#\boldsymbol{e}_j \setminus \widehat{\boldsymbol{y}_j} < \#\boldsymbol{e}_j$ since $\widehat{\boldsymbol{y}_j}$ is of size at least d_j), we know that this is also the case for every $\boldsymbol{x} \in \mathcal{P}_{\leq d_j}(\widehat{\boldsymbol{y}_j})$. Remains to prove that every subset of $\boldsymbol{e}_j$ of size d_j that has some elements in $\widehat{\boldsymbol{y}_j}$ and some elements outside of it is also independent from $\boldsymbol{x_H}$. This is dealt with by the for loop on lines 7-8, which covers all possible combinations to recombine $\boldsymbol{y}_j$ and its complement, in parallel for all j. □

Comparison. Both algorithms lead to significant improvements in the verification time compared to the naive method which enumerates all t-tuples of observations for a given implementation. Further, our divide-and-conquer strategies make feasible the verification of some masked programs on which enumeration is simply unfeasible. To illustrate both these improvements and the differences between our algorithms, we apply the three methods to the S-box of [13] (Algorithm 4) protected at various orders. Table 1 shows the results, where column *# tuples* contains the total number of tuples of program points to be considered, column *# sets* contains the number of sets used by the splitting algorithms and the *time* column shows the verification times when run on a headless VM with a dual core[5] 64-bit processor clocked at 2GHz.

As can be seen, the worklist-based method is generally the most efficient one. In the following, and in particular in Section 5, we use the check function from Algorithm 6.

Discussion. Note that in both Algorithms 5 and 6, the worst execution time occurs when the call to extend does not in fact increase the size of the observation set under study. In the unlikely event where this occurs in *all* recursive calls, both algorithms degrade into an exhaustive enumeration of all tuples, which is no worse than the naive implementation.

However, this observation makes it clear that it is important for the extend function to extend observation sets as much as possible. It could be interesting, and would definitely be valuable, to find a good balance between the complexity and precision of the extend function.

[5] Only one core is used in the computation.

Table 1. Comparison of Algorithms 5 and 6 with naive enumeration and with each other

Method	# tuples	Security	Complexity	
			# sets	time
First-Order Masking				
naive	63	✓	63	0.001s
pair	63	✓	17	0.001s
list	63	✓	17	0.001s
Second-Order Masking				
naive	12,561	✓	12,561	0.180s
pair	12,561	✓	851	0.046s
list	12,561	✓	619	0.029s
Third-Order Masking				
naive	4,499,950	✓	4,499,950	140.642s
pair	4,499,950	✓	68,492	9.923s
list	4,499,950	✓	33,075	3.894s
Fourth-Order Masking				
naive	2,277,036,685	✓	-	unpractical
pair	2,277,036,685	✓	8,852,144	2959.770s
list	2,277,036,685	✓	3,343,587	879.235s

5 Experiments

In this section, we aim to show on concrete examples the efficiency of the methods we considered so far. This evaluation is performed using a prototype implementation of our algorithms that uses the EasyCrypt [6] tool's internal representations of programs and expressions, and relying on some of its low-level tactics for substitution and conversion. As such, the prototype is not designed for performance, but rather for trust, and the time measurements given below could certainly be improved. However, the numbers of sets each algorithm considers are fixed by our choice of algorithm, and by the particular choose algorithms we decided to use. We detail and discuss this particular implementation decision at the end of this section.

Our choice of examples mainly focuses on higher-order masking schemes since they are much more promising than the schemes dedicated to small orders. Aside from the masking order itself, the most salient limiting factor for performance is the size of the program considered, which is also (more or less) the number of observations that need to be considered. Still, we analyze programs of sizes ranging from simple multiplication algorithms to either round-reduced or full AES, depending on the masking order.

We discuss our practical results depending on the leakage model considered: we first discuss our prototype's performance in the value-based leakage model, then focus on results obtained in the transition-based leakage model.

Table 2. Verification of state-of-the-art higher-order masking schemes with # tuples the number t-uples of the algorithm at order t, # sets the number of sets built by our prototype and time the verification time in seconds

Reference	Target	# tuples	Result	Complexity	
				# sets	time (s)
First-Order Masking					
CHES10 [31]	multiplication	13	secure ✓	7	ε
FSE13 [13]	Sbox (4)	63	secure ✓	17	ε
FSE13 [13]	full AES (4)	17,206	secure ✓	3,342	128
MAC-Keccak	full Keccak-f	13,466	secure ✓	5,421	405
Second-Order Masking					
RSA06 [33]	Sbox	1,188,111	secure ✓	4,104	1.649
CHES10 [31]	multiplication	435	secure ✓	92	0.001
CHES10 [31]	Sbox	7,140	1^{st}-order flaws (2)	866	0.045
CHES10 [31]	key schedule [13]	23,041,866	secure ✓	771,263	340,745
FSE13 [13]	AES 2 rounds (4)	25,429,146	secure ✓	511,865	1,295
FSE13 [13]	AES 4 rounds (4)	109,571,806	secure ✓	2,317,593	40,169
Third-Order Masking					
CHES10 [31]	multiplication	24,804	secure ✓	1,410	0.033
FSE13 [13]	Sbox(4)	4,499,950	secure ✓	33,075	3.894
FSE13 [13]	Sbox(5)	4,499,950	secure ✓	39,613	5.036
Fourth-Order Masking					
RSA06 [33]	Sbox	4,874,429,560	3^{rd}-order flaws (98, 176)	35,895,437	22,119
CHES10 [31]	multiplication	2,024,785	secure ✓	33,322	1.138
FSE13 [13]	Sbox (4)	2, 277, 036, 685	secure ✓	3,343,587	879
Fifth-Order Masking					
CHES10 [31]	multiplication	216,071,394	secure ✓	856,147	45

5.1 Value-Based Model

Table 2 lists the performance of our prototype on multiple examples, presenting the total number of sets of observations to be considered (giving an indication of each problem's relative difficulty), as well as the number of sets used to cover all tuples of observations by our prototype. We also list the verification time, although these could certainly be improved independently of the algorithms themselves. Each of our tests is identified by a reference and a function, with additional information where relevant. The MAC-Keccak example is a simple implementation of Keccak-f on 64-bit words, masked using a variant of Ishai, Sahai and Wagner's transformation [21,31] (noting that their SecMult algorithm can be used to securely compute any associative and commutative binary operation that distributes over field addition, including bitwise ANDs).

The two rows without checkmarks correspond to examples on which the tool fails to prove t-non-interference. We now analyze them in more detail.

On Schramm and Paar's table-based implementation of the AES Sbox, supposed to be secure at order 4, our tool finds 98,176 third-order observations that

it cannot prove independent from the secrets. The time listed is the time needed to cover all triples, and the first error is found in 0.221s. These errors in fact correspond to four families of observations, which we now describe. Denoting by $X = \bigoplus_{0 \leqslant i \leqslant 4} x_i$ the S-box input and by $Y = \bigoplus_{0 \leqslant i \leqslant 4} y_i$ its output, we can write the four sets of flawed triples as follows:

1. $(x_0, \mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus (Y \oplus y_0), \mathsf{Sbox}(X \oplus x_0 \oplus j) \oplus (Y \oplus y_0))$, $\forall i, j \in \mathsf{GF}(2^8), i \neq j$
2. $(y_0, \mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus (Y \oplus y_0), \mathsf{Sbox}(X \oplus x_0 \oplus j) \oplus (Y \oplus y_0))$, $\forall i, j \in \mathsf{GF}(2^8),\ i \neq j$
3. $(x_0, \mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus (Y \oplus y_0 \oplus y_4), \mathsf{Sbox}(X \oplus x_0 \oplus j) \oplus (Y \oplus y_0 \oplus y_4))$, $\forall i, j \in \mathsf{GF}(2^8),\ i \neq j$
4. $(x_0, y_0, \mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus (Y \oplus y_0))$, $\forall i \in \mathsf{GF}(2^8)$.

We recall that y_0 is read as $y_0 = \mathsf{Sbox}(x_0)$, and prove that all four families of observations in fact correspond to attacks.

The first family corresponds to the attack detailed by Coron, Prouff and Rivain [12]). By summing the second and third variables, the attacker obtains $\mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus \mathsf{Sbox}(X \oplus x_0 \oplus j)$. The additional knowledge of x_0 clearly breaks the independence from X. To recover secrets from a second set's triple of observations, the attacker can sum the second and third variables to obtain $X \oplus x_0$, from which he can learn $Y \oplus y_0$ (by combining it with the second variable) and then Y (by combining it with the first one). The third family is a variant of the first: the S-box masks can be removed in both cases. Finally, when observing three variables in the fourth family of observations, the knowledge of both x_0 and y_0 unmasks the third observed variable, making it dependent on X.

Our tool also finds two suspicious adversary observations on the S-box algorithm proposed by Rivain and Prouff [31], that in fact correspond to the two flaws revealed in [13]. However, by the soundness of our algorithm, and since our implementation only reports these two flaws, we now know that these are the only two observations that reveal any information on the secrets. We consider several corrected versions of this S-box algorithm, listed in Table 3. Some of these fixes focused on using a more secure mask refreshing function (borrowed from [14]) or refreshing all modified variables that are reused later on (as suggested by [30]). Others make use of specialized versions of the multiplication algorithm [13] that allow the masked program to retain its performance whilst gaining in security.

Although it is important to note that the algorithms appear to be "precise enough" in practice, Table 2 also reveals that program size is not in fact the only source of complexity. Indeed, proving the full key schedule at order 2 only involves around 23 million pairs of observations, compared to the 109 million that need to be considered to prove the security of 4 rounds of AES at the same order; yet the latter takes less than an hour to complete compared to 4 days for the full ten rounds of key schedule. We suspect that this is due to the shapes of the two programs' dependency graphs, with each variable in the key schedule depending on a large proportion of the program's input variables, whereas the dependencies in full AES are sparser. Although properties of composition would

Table 3. Fixing RP-CHES10 [31] at the second order

Reference	S-box	# tuples	Result	Complexity # sets	Complexity time
Second-Order Masking					
RP-CHES10 [31]	initially proposed	7,140	1^{st}-order flaws (2)	840	0.070s
RP-CHES10 [31]	different refreshMasks	7,875	secure ✓	949	0.164s
RP-CHES10 [31]	more refreshMasks	8,646	secure ✓	902	0.180s
CPRR-FSE13 [13]	use of $x \cdot g(x)$ (Algo 4)	12,561	secure ✓	619	0.073s
CPRR-FSE13 [13]	use of tables (Algo 5)	12,561	secure ✓	955	0.196s

allow us to consider large programs masked at much higher orders, we leave these investigations to further works.

Another important factor in the performance of our algorithm is the instantiation of the various choice functions. We describe them here for the sake of reproducibility. In Algorithms 1 and 2, when choosing a triple $(\boldsymbol{e'}, e, r)$ to use with rule (Opt), our prototype first chooses r as the first (leftmost-first depth-first) random variable that fulfills the required conditions, then chooses e as the *largest* superterm of r that fulfills the required conditions (this fixes $\boldsymbol{e'}$). When choosing an expression to observe (in Algorithms 5 and 6) or to extend a set of observation with (in Algorithm 4), we choose first the expression that has the highest number of dependencies on random or input variables. These decisions certainly may have a significant effect on our algorithm's performance, and investigating these effects more deeply may help gather some insight on the core problems related to masking. We leave this a future work.

5.2 Transition-Based Model

The value-based leakage model may not always be the best fit to capture the behaviour of hardware and software. In particular, when considering software implementations, it is possible that writing a value into a register leaks both its new and old contents. To illustrate the adaptability of our algorithms, we first run some simple tests. We then illustrate another potential application of our tool, whereby masked implementations that make use of $t+1$ masks per variable can be proved secure in the transitions model at orders much higher than the generic $t/2$, simply by reordering instructions and reallocating registers.

Table 4 describes the result of our experiments. Our first (naive) implementation is only secure at the second order in the transition-based leakage model and uses 21 local registers (the number of registers needed for this and other implementations to be secure could also be reduced further by zeroing out registers between independent uses). Our first improved implementation achieves security at order 3 in the transition-based leakage model with only 6 local registers. Trying to provide the best possible security in this model, we also find a third implementation that achieves security at order 4. This last implementation is in fact the original implementation with additional registers. Note however,

Table 4. Multiplication in the transition-based leakage model

Reference	Multiplication	# tuples	Security	Complexity	
				# sets	time
RP-CHES10 [31]	initial scheme for order 4	3,570	order 2	161	0.008s
RP-CHES10 [31]	with some instructions reordering	98,770	order 3	3,488	0.179s
RP-CHES10 [31]	using more registers	2,024,785	order 4	17,319	1.235s

that in spite of its maximal security order, this last implementation still reuses registers (in fact, most are used at least twice).

The main point of these experiments is to show that the techniques and tools we developed are helpful in building and verifying implementations in other models. Concretely, our tools give countermeasure designers the chance to easily check the security of their implementation in one or the other leakage model, and identify problematic observations that would prevent the countermeasure from operating properly against higher-order adversaries.

6 Conclusion

This paper initiates the study of relational verification techniques for checking the security of masked implementations against t-order DPA attacks. Beyond demonstrating the feasibility of this approach for masking orders higher than 2, our work opens a number of interesting perspectives on automated DPA tools.

The most immediate direction for further work is to exhibit and prove compositional properties in order to achieve the verification of larger masked programs at higher orders.

Another promising direction is to automatically synthesize efficient and secure implementations by search-based optimization. Specifically, we envision a 2-step approach where one first uses an unoptimized but provably secure compiler to transform a program p into a program $\overline{p}_t$ that is t-non-interfering, and then applies relational synthesis methods, in the spirit of [4], to derive a more efficient program p' that is observationally equivalent to $\overline{p}_t$ and equally secure—the latter property being verified using pRHL.

Acknowledgments. We thank F.-X. Standaert and V. Grosso for giving us access to their examples and code generation tools, and H. Eldib and C. Wang for letting us make use of their code samples for comparison purposes. This research is partially funded by Spanish projects TIN2009-14599 DESAFIOS 10 and TIN2012-39391-C04-01 StrongSoft, Madrid Regional project S2009TIC-1465 PROMETIDOS, ANR project ANR-14-CE28-0015 BRUTUS and ANR project ANR-10-SEGI-015 PRINCE.

A Initial Transformations on Programs: An Example

To illustrate our algorithms, we consider the simple masked multiplication algorithm defined in [31] and relying on Algorithm 7, which is secure against 2-threshold probing adversaries. In practice, the code we consider is in 3-address

form, with a single operation per line (operator application or table lookup). For brevity, we use parentheses instead, unless relevant to the discussion. In the rest of this paper, we write Line $(n).i$ to denote the i^{th} expression computed on line n, using the convention that products are computed immediately before their use. For example, Line (5).1 is the expression $a_0 \odot b_1$, Line (5).2 is $r_{0,1} \oplus a_0 \odot b_1$ and Line (5).3 is $a_1 \odot b_0$.

Algorithm 7. Secure Multiplication Algorithm ($t = 2$) from [31]

Input: a_0, a_1, a_2 (resp. b_0, b_1, b_2) such that $a_0 \oplus a_1 \oplus a_2 = a$ (resp. $b_0 \oplus b_1 \oplus b_2 = b$)
Output: c_0, c_1, c_2 such that $c_0 \oplus c_1 \oplus c_2 = a \odot b$

1: **function** SecMult($[\![a_0, a_1, a_2]\!], [\![b_0, b_1, b_2]\!]$)
2: $r_{0,1} \xleftarrow{\$} \mathbb{F}_{256}$
3: $r_{0,2} \xleftarrow{\$} \mathbb{F}_{256}$
4: $r_{1,2} \xleftarrow{\$} \mathbb{F}_{256}$
5: $r_{1,0} \leftarrow (r_{0,1} \oplus a_0 \odot b_1) \oplus a_1 \odot b_0$
6: $r_{2,0} \leftarrow (r_{0,2} \oplus a_0 \odot b_2) \oplus a_2 \odot b_0$
7: $r_{2,1} \leftarrow (r_{1,2} \oplus a_1 \odot b_2) \oplus a_2 \odot b_1$
8: $c_0 \leftarrow (a_0 \odot b_0 \oplus r_{0,1}) \oplus r_{0,2}$
9: $c_1 \leftarrow (a_1 \odot b_1 \oplus r_{1,0}) \oplus r_{1,2}$
10: $c_2 \leftarrow (a_2 \odot b_2 \oplus r_{2,0}) \oplus r_{2,1}$
11: **return** $[\![c_0, c_1, c_2]\!]$

Algorithm 8. Presharing, Sharing and Preprocessed multiplication ($t = 2$, a is secret, b is public)

1: **function** PreShare(a)
2: $a_0 \xleftarrow{\$} \mathbb{F}_{256}$
3: $a_1 \xleftarrow{\$} \mathbb{F}_{256}$
4: $a_2 \leftarrow [a \oplus a_0 \oplus a_1]$
5: **return** $[\![a_0, a_1, a_2]\!]$

1: **function** Share(a)
2: $a_0 \xleftarrow{\$} \mathbb{F}_{256}$
3: $a_1 \xleftarrow{\$} \mathbb{F}_{256}$
4: $a_2 \leftarrow (a \oplus a_0) \oplus a_1$
5: **return** $[\![a_0, a_1, a_2]\!]$

1: **function** $\overline{\text{SecMult}}(a, b)$
2: $a_0 \xleftarrow{\$} \mathbb{F}_{256}$
3: $a_1 \xleftarrow{\$} \mathbb{F}_{256}$
4: $a_2 \leftarrow [a \oplus a_0 \oplus a_1]$
5: $b_0 \xleftarrow{\$} \mathbb{F}_{256}$
6: $b_1 \xleftarrow{\$} \mathbb{F}_{256}$
7: $b_2 \leftarrow (b \oplus b_0) \oplus b_1$
8: $r_{0,1} \xleftarrow{\$} \mathbb{F}_{256}$
9: $r_{0,2} \xleftarrow{\$} \mathbb{F}_{256}$
10: $r_{1,2} \xleftarrow{\$} \mathbb{F}_{256}$
11: $r_{1,0} \leftarrow (r_{0,1} \oplus a_0 \odot b_1) \oplus a_1 \odot b_0$
12: $r_{2,0} \leftarrow (r_{0,2} \oplus a_0 \odot b_2) \oplus a_2 \odot b_0$
13: $r_{2,1} \leftarrow (r_{1,2} \oplus a_1 \odot b_2) \oplus a_2 \odot b_1$
14: $c_0 \leftarrow (a_0 \odot b_0 \oplus r_{0,1}) \oplus r_{0,2}$
15: $c_1 \leftarrow (a_1 \odot b_1 \oplus r_{1,0}) \oplus r_{1,2}$
16: $c_2 \leftarrow (a_2 \odot b_2 \oplus r_{2,0}) \oplus r_{2,1}$
17: **return** $[c_0 \oplus c_1 \oplus c_2]$

Line	Observed Expression	Line	Observed Expression
(2)	a_0	(12).2	$r_{0,2} \oplus a_0 \odot b_2$
(3)	a_1	(12).3	$a_2 \odot b_0$
(4)	$a_2 := (a \oplus a_0) \oplus a_1$	(12)	$(r_{0,2} \oplus a_0 \odot b_2) \oplus a_2 \odot b_0$
(5)	b_0	(13).1	$a_1 \odot b_2$
(6)	b_1	(13).2	$r_{1,2} \oplus a_1 \odot b_2$
(7).1	$b \oplus b_0$	(13).3	$a_2 \odot b_1$
(7)	$b_2 := (b \oplus b_0) \oplus b_1$	(13)	$(r_{1,2} \oplus a_1 \odot b_2) \oplus a_2 \odot b_1$
(8)	$r_{0,1}$	(14).1	$a_0 \odot b_0$
(9)	$r_{0,2}$	(14).2	$a_0 \odot b_0 \oplus r_{0,1}$
(10)	$r_{1,2}$	(14)	$(a_0 \odot b_0 \oplus r_{0,1}) \oplus r_{0,2}$
(11).1	$a_0 \odot b_1$	(15).1	$a_1 \odot b_1$
(11).2	$r_{0,1} \oplus a_0 \odot b_1$	(15).2	$a_1 \odot b_1 \oplus ((r_{0,1} \oplus a_0 \odot b_1) \oplus a_1 \odot b_0)$
(11).3	$a_1 \odot b_0$	(15)	$(a_1 \odot b_1 \oplus ((r_{0,1} \oplus a_0 \odot b_1) \oplus a_1 \odot b_0)) \oplus r_{1,2}$
(11)	$(r_{0,1} \oplus a_0 \odot b_1) \oplus a_1 \odot b_0$	(16).1	$a_2 \odot b_2$
(12).1	$a_0 \odot b_2$	(16).2	$a_2 \odot b_2 \oplus ((r_{0,2} \oplus a_0 \odot b_2) \oplus a_2 \odot b_0)$
		(16)	(16).2 $\oplus((r_{1,2} \oplus a_1 \odot b_2) \oplus a_2 \odot b_1)$

Fig. 2. Possible wire observations for $\overline{\text{SecMult}}$. (Note that, after Lines 4 and 7, we keep a_2 and b_2 in expressions due to margin constraints.)

When given a program whose inputs have been annotated as secret or public, we transform it as described at the end of Section 2.4 to add some simple initialization code that preshares secrets in a way that is not observable by the adversary, and lets the adversary observe the initial sharing of public inputs. This allows us to model, as part of the program, the assumption that shares of the secret are initially uniformly distributed and that their sum is the secret. The initialization code, as well as the transformed version of Algorithm 7 where argument a is marked as secret and b is marked as public, are shown in Algorithm 8. We use the square brackets on Line (4) of function PreShare to mean that the intermediate results obtained during the computation of the bracketed expression are not observable by the adversary: this is equivalent to the usual assumption that secret inputs and state are shared before the adversary starts performing measurements.

Once the program is in this form, it can be transformed to obtain: i. the set of its random variables;[6] ii. the set of expressions representing all of the possible adversary observations; This final processing step on $\overline{\text{SecMult}}$ yields the set of random variables $\mathcal{R} = \{a_0, a_1, b_0, b_1, r_{0,1}, r_{0,2}, r_{1,2}\}$, and the set of expressions shown in Figure 2 (labelled with their extended line number). Recall that these sets were obtained with a marked as secret and b marked as public.

[6] In practice, since we consider programs in SSA form, it is not possible to assign a non-random value to a variable that was initialized with a random.

Line	Register	Old Contents	New Contents
(2)		$\perp$	a_0
(3)		$\perp$	a_1
(4)		$\perp$	$a \oplus a_0 \oplus a_1$
(5)		$\perp$	b_0
(6)		$\perp$	b_1
(7).1	b_2	$\perp$	$b \oplus b_0$
(7)		$b \oplus b_0$	$b \oplus b_0 \oplus a_1$
(8)		$\perp$	$r_{0,1}$
(9)		$\perp$	$r_{0,2}$
(10)		$\perp$	$r_{1,2}$
(11).1	$r_{1,0}$	$\perp$	$a_0 \odot b_1$
(11).2	$r_{1,0}$	$a_0 \odot b_1$	$r_{0,1} \oplus a_0 \odot b_1$
(11).3	t	$\perp$	$a_1 \odot b_0$
(11)		$r_{0,1} \oplus a_0 \odot b_1$	$r_{0,1} \oplus a_0 \odot b_1 \oplus a_1 \odot b_0$
(12).1	$r_{2,0}$	$\perp$	$a_0 \odot b_2$
(12).2	$r_{2,0}$	$a_0 \odot b_2$	$r_{0,2} \oplus a_0 \odot b_2$
(12).3	t	$a_1 \odot b_0$	$a_2 \odot b_0$
(12)		$r_{0,2} \oplus a_0 \odot b_2$	$r_{0,2} \oplus a_0 \odot b_2 \oplus a_2 \odot b_0$
(13).1	$r_{2,1}$	$\perp$	$a_1 \odot b_2$
(13).2	$r_{2,1}$	$a_1 \odot b_2$	$r_{1,2} \oplus a_1 \odot b_2$
(13).3	t	$a_2 \odot b_0$	$a_2 \odot b_1$
(13)		$r_{1,2} \oplus a_1 \odot b_2$	$r_{1,2} \oplus a_1 \odot b_2 \oplus a_2 \odot b_1$
(14).1	c_0	$\perp$	$a_0 \odot b_0$
(14).2	c_0	$a_0 \odot b_0$	$a_0 \odot b_0 \oplus r_{0,1}$
(14)		$a_0 \odot b_0 \oplus r_{0,1}$	$a_0 \odot b_0 \oplus r_{0,1} \oplus r_{0,2}$
(15).1	c_1	$\perp$	$a_1 \odot b_1$
(15).2	c_1	$a_1 \odot b_1$	$a_1 \odot b_1 \oplus r_{0,1} \oplus a_0 \odot b_1 \oplus a_1 \odot b_0$
(15)		$a_1 \odot b_1 \oplus r_{0,1} \oplus a_0 \odot b_1 \oplus a_1 \odot b_0$	(15).2 $\oplus r_{1,2}$
(16).1	c_2	$\perp$	$a_2 \odot b_2$
(16).2	c_2	$a_2 \odot b_2$	$a_2 \odot b_2 \oplus r_{0,2} \oplus a_0 \odot b_2 \oplus a_2 \odot b_0$
(16)		$a_2 \odot b_2 \oplus r_{0,2} \oplus a_0 \odot b_2 \oplus a_2 \odot b_0$	(16).2 $\oplus r_{1,2} \oplus a_1 \odot b_2 \oplus a_2 \odot b_1$

Fig. 3. Possible transition observations for $\overline{\text{SecMult}}$ with a naive register allocation (shown in the last column). $\perp$ denotes an uninitialized register, whose content may already be known to (and perhaps chosen by) the adversary.

A.1 Observable Transitions

Figure 3 presents the observable transitions for Algorithm 7. It gives the old value and the new value of the register modified by each program point. This is done using a simple register allocation of Algorithm 7 (where we use the word "register" loosely, to denote program variables, plus perhaps some additional temporary registers if required) that uses a single temporary register that is never cleared, and stores intermediate computations in the variable where their end result is stored. For clarity, the register in which the intermediate result is stored is also listed in the Figure.

References

1. Akinyele, J., Barthe, G., Grégoire, B., Schmidt, B., Strub, P.-Y.: Certified synthesis of efficient batch verifiers. In: 27th IEEE Computer Security Foundations Symposium, CSF 2014. IEEE Computer Society (2014) (to appear)
2. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.-X.: On the cost of lazy engineering for masked software implementations. Cryptology ePrint Archive, Report 2014/413 (2014). http://eprint.iacr.org/2014/413
3. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., Strub, P.-Y.: Verified proofs of higher-order masking. Cryptology ePrint Archive, Report 2015/060 (2015). http://eprint.iacr.org/
4. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to SIMD loop synthesis. In: Nicolau, A., Shen, X., Amarasinghe, S.P., Vuduc, R.W. (eds.) Principles and Practice of Parallel Programming (PPoPP), pp. 123–134. ACM (2013)
5. Barthe, G., Daubignard, M., Kapron, B., Lakhnech, Y., Laporte, V.: On the equality of probabilistic terms. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 46–63. Springer, Heidelberg (2010)
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.-Y.: EasyCrypt: a tutorial. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) FOSAD VII. LNCS, vol. 8604, pp. 146–166. Springer, Heidelberg (2014)
7. Barthe, G., Grégoire, B., Heraud, S., Zanella-Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
8. Barthe, G., Grégoire, B., Zanella-Béguelin, S.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 90–101. ACM (2009)
9. Bayrak, A.G., Regazzoni, F., Novo, D., Ienne, P.: Sleuth: automated verification of software power analysis countermeasures. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 293–310. Springer, Heidelberg (2013)
10. Canright, D., Batina, L.: A very compact "perfectly masked" S-box for AES. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 446–459. Springer, Heidelberg (2008)
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
12. Coron, J.-S., Prouff, E., Rivain, M.: Side channel cryptanalysis of a higher order masking scheme. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 28–44. Springer, Heidelberg (2007)
13. Coron, J.-S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 410–424. Springer, Heidelberg (2014)
14. Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: from probing attacks to noisy leakage. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 423–440. Springer, Heidelberg (2014)
15. Eldib, H., Wang, C.: Synthesis of masking countermeasures against side channel attacks. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 114–130. Springer, Heidelberg (2014)
16. Eldib, H., Wang, C., Schaumont, P.: SMT-based verification of software countermeasures against side-channel attacks. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 62–77. Springer, Heidelberg (2014)

17. Eldib, H., Wang, C., Taha, M.M.I., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code. In: The 51st Annual Design Automation Conference 2014, DAC 2014, San Francisco, CA, USA, June 1–5, pp. 1–6. ACM (2014)
18. Faust, S., Rabin, T., Reyzin, L., Tromer, E., Vaikuntanathan, V.: Protecting circuits from leakage: the computationally-bounded and noisy cases. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 135–156. Springer, Heidelberg (2010)
19. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
20. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 633–654. IOS Press (2009)
21. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)
22. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
23. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
24. Micali, S., Reyzin, L.: Physically observable cryptography (extended abstract). In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 278–296. Springer, Heidelberg (2004)
25. Moss, A., Oswald, E., Page, D., Tunstall, M.: Compiler assisted masking. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 58–75. Springer, Heidelberg (2012)
26. Oswald, E., Mangard, S.: Template attacks on masking—resistance is futile. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 243–256. Springer, Heidelberg (2007)
27. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A side-channel analysis resistant description of the AES S-box. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 413–423. Springer, Heidelberg (2005)
28. Peeters, E., Standaert, F.-X., Donckers, N., Quisquater, J.-J.: Improved higher-order side-channel attacks with FPGA experiments. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 309–323. Springer, Heidelberg (2005)
29. Pettai, M., Laud, P.: Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. Cryptology ePrint Archive, Report 2014/240 (2014). http://eprint.iacr.org/2014/240
30. Prouff, E., Rivain, M.: Masking against side-channel attacks: a formal security proof. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 142–159. Springer, Heidelberg (2013)
31. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
32. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
33. Schramm, K., Paar, C.: Higher order masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006)
34. Veyrat-Charvillon, N., Gérard, B., Standaert, F.-X.: Soft analytical side-channel attacks. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, PART I. LNCS, vol. 8873, pp. 282–296. Springer, Heidelberg (2014)