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
{gilles.barthe,francois.dupressoir,pierre-yves.strub}@imdea.org
[2] École normale supérieure and Thales Communications & Security
sonia.belaid@ens.fr
[3] Université de Rennes 1 and Institut universitaire de France
pierre-alain.fouque@ens.fr
[4] INRIA
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 (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 type systems have been proposed to help cryptographers check their proofs, but many of these approaches are insufficient for higher-order implementations.
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: Masking, Automatic tools, EasyCrypt, Higher-Order Masking

1 Introduction

This paper discusses the issue of automatically verifying the security of higher-order masked implementations, which are gaining in importance since differential power or electromagnetic analysis [31] are the most efficient attacks on block-cipher implementations. For instance, recent and very efficient attacks have allowed adversaries to recover the secret key using only one AES power trace [49]. Such attacks are making the use of masking, a well-known and effective countermeasure to protect against side-channel attacks, increasingly popular. Masking consists in using a secret-sharing scheme to split each sensitive variable into $(t + 1)$ shares such that the joint distribution of any subset of at most t shares is uniform and 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. In practice, even higher-order attacks, where t is greater than 1 have been conducted [37,39], and therefore deserve to be considered when designing countermeasures. The increasing number of masking schemes and masked implementations that were proved secure and later found vulnerable to attacks make them an ideal target for formal methods and proof checking.

Formal Security and Leakage Models. Chari et al. [14] performed the first formal security analysis of masking, by showing that the number of queries needed to recover a sensitive bit is at least exponential in the masking order t in a *noisy leakage model*. In this model, the adversary does not know the exact value of internal variables, but only access noisy leakage channels. More precisely, the adversary gets a leaked value sampled according to a gaussian distribution centered around the actual value. This model is an abstraction of the real information gained by the adversary since, for example when attacking smartcards, one can only observe the noisy Hamming weight of a variable, or the Hamming distance between two successive values of a variable or register. In other side-channel attacks such as cache attacks, one can obtain the most significant bits of a byte used as index in a table lookup. In this case, the information obtained is close to the noisy leakage model. Following [14], Ishai, Sahai and Wagner [29] proposed another leakage model, the *t-threshold probing model*, where the adversary knows the exact value of *at most t internal variables* in the computation. This model was originally inspired from chip attacks described for instance in [2]. Ishai, Sahai and Wagned [29] also provide a compiler that transforms any circuit C secure in the *black-box model* into a circuit C' secure in the t-threshold probing model. Many other leakage models have been considered in the literature since these papers [34,19,48,23]. In practice, the noisy leakage model is often thought of as more realistic, since experimental physical leakages are noisy [33]. It is worth noting that, although the t-threshold probing model enables the adversary to observe exact values rather than noisy ones, the model is not clearly more powerful than the noisy leakage model, since the two models also vary in the number of observations the adversary is allowed.

AES masking and Problems in proofs. Since AES implementations are preferred targets for side-channel attacks, we focus on masked implementations of AES and its components. Many papers have proposed masked implementations of AES or only of its non-linear component, the S-box [11,38,12,47,44,25,45,43,30,27,13,42]. Some of these masked implementations were 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 seems to be a more difficult task and many errors appeared in the literature. For instance, the schemes presented by [47] and [45] were later broken in [16] and [17]. Consequently, it is important to formally verify masked implementations and many tools to do so appeared recently [35,10,21,20,22]. The first tools [35,10] use type systems to propagate sensitivity marks along the programs. However as clearly explained in [21], such approaches are not complete and many programs are incorrectly evaluated. The main advantage of type systems is that the tools they yield are very efficient. However, they cannot check properties on probability distributions. Other tools have been proposed to take into account distributions, but appear to only be able to handle small circuits, and it is not obvious that results obtained on small programs by those tools can be composed or combined to obtain security results on larger ones.

Noisy and Probing Models. Prouff and Rivain [42] extend the noisy model of Chari et al. [14] to incorporate more general probability leakage distributions rather than just gaussian [14] or Bernoulli leakage [23]. Moreover, they remove the limitation to one-bit observations, allowing the adversary to observe any intermediate variable, and also take the computation into account, while [14] only focuses on shares independently of the computation. The model follows the *only computation leaks information* model introduced by Micali and Reyzin in [34]. At each computation step, also called *elementary calculation* or CPU instruction in practice, one can learn a leakage function $f(x)$ that depends only on part of the state x. In [42], the noise function is modeled by bounding the bias in the distribution of x given $f(x)$, that is the statistical distance between the distribution $\Pr[x]$ and $\Pr[x|f(x)]$ is bounded depending on some security parameter ω. They prove the security of a block cipher implementation in this model by bounding the mutual information on the key and the plaintext given the leakage functions as $\omega^{-(t+1)}$, with t the masking order. Prouff and Rivain's work in [42] is the first proof of security for a block cipher in such a practical and low-level security model. However, the notion of security used is information theoretic and cannot be used directly in more standard reductionist security proofs. Consequently, for their result to hold, the adversary cannot know the ciphertext since this information allows to determine the key given a few plaintext / ciphertext pairs. Recently, Duc, Dziembowski and Faust [18] recast the noisy leakage in the more classic statistical security model and show that the security in the noisy leakage model of [42] can be reduced to the security in the t-threshold probing model of [29], in which security proofs are much more direct. In addition, the adversary model in [18] captures chosen plaintext attacks, and the proof does not rely on the existence of leak-free components.

Transition-based Model. In the noisy leakage model as well as in the t-threshold probing model, only the values of intermediate variables are considered when determining the security order of an implementation. However, Balash et al. [3] show that this value-based leakage modelling does not fully capture some real-world scenarios, where additional leakage may come from transitions at the gate level called *glitches*. As a consequence, a perfectly masked algorithm secure in the value-based model can face first-order flaws based on its transitions. In the software scenario, glitches are replaced by memory transitions, where an arbitrary function (usually the Hamming distance) of the old and new value of a register may be leaked during the register write, in a single observation. As explained in [3], the consequence of these practical issues is (at worst) a division of the security order by two.

Our Approach. We rephrase security in the t-threshold probing model as the notion of *t-non interference*, based on non-interference notions used for verifying information-flow properties in language-based security. We first define a more 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 conditional probability distributions on $\mathcal{O}$ defined by p_1 and p_2 are equal, given the assumptions on input variables encoded in $\mathcal{I}$.

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 and a set of *observed variables* $\mathcal{O}$ when $\bar{p}(s_0, \cdot)$ and $\bar{p}(s_1, \cdot)$ are $(\mathcal{I}, \mathcal{O})$-equivalent (with $\mathcal{I}$ the set of non-secret input variables) for any value of the secret input variables s_0 and s_1.

We now give an indistinguishability-based definition of the t-threshold probing model. In our 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: computation always uses secrets s_0, but the leaks given to the adversary are produced 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 [18] and Ishai, Sahai and Wagner [29].

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.

Proof. Since p and $\overline{p}$ are functionally equivalent, we have $\Delta(\mathcal{S} \overset{bb}{\leftrightarrows} p, \mathcal{S} \overset{bb}{\leftrightarrows} \overline{p}) = 0$ and we just have to prove that $\Delta(\mathcal{S} \overset{bb}{\leftrightarrows} \overline{p}, \mathcal{A} \overset{thr}{\leftrightarrows} \overline{p}) = 0$. 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 $\overline{p}$, he can choose arbitrary secret variables, run $\overline{p}$ with these variables and the public variables given as inputs, and outputs the observations. □

The theorem can be lifted to the noisy leakage model using Corollary 1 from [18], using a small bound on the statistical distance instead.

Following Theorem 1, we propose algorithms to prove functional equivalence 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.

We have evaluated the practicality of our approach by implementing our algorithms on top of EasyCrypt [7,6], and testing the performance of our implementation on representative examples from the literature. 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 on flawed implementations, to check that proposed fixes, when they exist, are indeed secure, and finally to discover new attacks on flawed implementations [47].

Our Results. The problem of verifying the security of higher-order masked implementation or finding attacks against such implementation is inherently exponential since it involves considering every combination of t out of n internal variables. More precisely, to prove t-non-interference, we show that the joint probability distribution on any t intermediate variables is independent from the secret inputs. We first review previous uses of formal methods to prove similar properties (Section 2). In Sections 3 and 4, we describe algorithms that greatly decrease, in practice, the number of checks that need to be performed. One idea consists in building sets, whose cardinalities are larger than t, of variables that are all independent. Consequently, we can take into account many subsets of size t without having to individually check each of them. Moreover, we propose a technique for systematically verifying that the masked implementation is functionally equivalent to a reference unprotected implementation. We implement all our algorithms and illustrate their efficiency for verifying the security of many schemes or part of AES masked implementations proposed in the literature in section 6. We recover previous attacks [16,17] and we discover new attacks on flawed implementations [47] by exhibiting observations sets that are not independent of the secret. Finally, we consider other leakage models such as the recent transition-based model and we experimentally show that implementations may be made resistant to a much higher-order than the generic results suggest.

Limitations of our approach. Our work deliberately focuses 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. The common strategy to address scalability issues is to develop compositional techniques. This is done, for instance, in the context of masking compilers, whose proof of security proceeds by showing that each gadget is secure, and that gadgets are combined securely. In this light, our algorithmic methods are primarily focused on proving that gadgets are secure.

2 Language-based techniques for threshold security in the probing model

This section provides 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 standards problems in programming languages, namely information flow checking and equivalence checking, which formalize the assumptions of Theorem 1. Next, we discuss two possible approaches for addressing these problems; the approaches are inspired from certified compilation and translation validation, and we motivate the use of the latter. 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 here we consider probabilistic programs. Information flow checking is a standard problem in language-based security, although it usually considers flows from secret inputs to public outputs—whereas here we consider flows from secret inputs to intermediate values.

Both problems can be construed as instances of relational verification problems. For the clarity of exposition, we formalize this view in the simple case of straightline code 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 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 and ℓ and ℓ' respectively denote the sizes of $\mathsf{IVar}(p)$ and $\mathsf{PVar}(p)$ and $\mathsf{OVar}(p)$. The function $[\![p]\!]$ takes as input a joint distribution on input variables and returns a joint distributions 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 every $\boldsymbol{v} \in \mathcal{V}^{\kappa}$ 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 $\Phi \subseteq \mathcal{D}(\mathcal{V}^{\kappa}) \times \mathcal{D}(\mathcal{V}^{\kappa})$, 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 $\mathsf{PVar}(p)$, i.e. $\mathbb{O} \subseteq \mathcal{P}(\mathsf{PVar}(p))$; 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-defined and low equivalent.

There is a direct interpretation of t-threshold probing security in terms of 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. 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 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.

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 $p_1 \sim p_2$, iff $[\![p_1]\!]_{\mathcal{W}}(\mu) = [\![p_2]\!]_{\mathcal{W}}(\mu)$ for every distribution μ such that $R\ \mu\ \mu$.

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 $p_1 \sim_{\Phi}^{\mathcal{O}} p_2$, 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 $p_1 \sim_{\Phi}^{\mathcal{O}} p_2$.

2.2 Certified compilation vs certifying compilation

There are two possible approaches to proving the validity of masking countermeasures. The first approach is inspired from certified compilation [32]. This approach either considers fixed-level transformations that take as input a program p and return a transformed program $\overline{p}$ intended to be secure at some fixed order t, or generic transformations that take as input a program p and return for all levels t a transformed program $\overline{p}_t$ intended to be secure at order t. The goal of this approach is to prove formally that for all programs p, all levels t and all transformed programs $\overline{p}_t$, the programs p and p_t are observationally equivalent and moreover the program $\overline{p}_t$ is t-non-interfering. In order to make the proof “formal”, certified compilation advocates using general purpose verification tools known as proof assistants; typically, the transformation is programmed, and proved correct, within the proof assistant—because correctness is expressed in terms of program semantics, it also entails that the semantics of programs is also modelled in the proof assistant. On the positive side, the proof is done once and for all, for all programs p and all levels t; moreover, the proof is compositional, in the sense that it follows the construction of $\overline{p}_t$ from p. On the negative side, the proof must be repeated for each transformation, and does not give guarantees about programs that are not obtained by applying the transformation.

The second approach is inspired from translation validation [41], also known as certifying compilation [36]. In this approach, one is given an original program p, a transformed program $\overline{p}$ and a target level t. The goal is to check automatically that p and $\overline{p}$ are equivalent and that $\overline{p}$ is non-interfering; in order to achieve efficient verification, translation validation provisions the possibility to provide additional inputs, called witnesses, that help to verify the desired property; typical examples of witnesses include type annotations or loop invariants. Compared to approaches based on certified compilation, approaches based on translation validation do not deliver generic guarantees that are typically established in theoretical papers on masking. On the other hand, these methods are very flexible, easier to put into practice, and do not make any assumption on the way that $\overline{p}$ is generated. Therefore, we opt to follow an approach based on translation validation. However, several technical ingredients of our approach provide an excellent basis for developing an approach inspired from certified compilation—provided, of course, that they would be formalized in a proof assistant.

2.3 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 (for instance, 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 [46] 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. [35] pioneered the application of information flow type system 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 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 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 [35]. 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 [35] 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.

2.4 Model counting

Model counting [28] is a generalization of the satisfiability problem, that aims to compute the number of valid assignments for a logical formula in a first-order theory. It is well-known that model counting can be used to prove equivalence of randomized algorithms that sample values uniformly from finite sets; this is formalized for instance in [7]. Building on this observation, Eldib, Wang and Schaumont [21] propose a SMT-based approach to verify that implementations are secure in the t-threshold probing model. Later, Eldib and Wang [20] elaborate on this approach and develop an algorithm for automatically synthesizing masked implementations. Unfortunately, their approach is based on a naive encoding of model counting and hence it can only be applied to lower masking orders, i.e. $t = 1$ or $t = 2$. An independent work by Fredrikson and Jha [24] develop a more efficient approach to model counting based on Barvinok's algorithm [9] and apply their approach to differential privacy.

2.5 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, i.e. 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-Béguelin [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 values 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 the clarity of 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 the 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 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 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 an automorphism on the domains of the distribution from which values are drawn, and proving that a formula derive 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.

It is important to 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, and fully observable code that simply shares public inputs. The code for sharing and presharing, as well as an example of this transformation are given in Appendix A.

2.6 Discussion

Both approaches based on type systems and relational verification can be used to check non-interference with respect to a single set of observations. However, these approaches are not practical for checking security in the t-threshold probing model, because the set of observation sets consists of all the possible subsets of program variables with size less than t, and hence grows exponentially in the size of the implementation, and hence in the order t for which we want to prove security. Therefore, more efficient methods need to be developed. One main technical contribution of this paper, detailed in Section 4 is a set of algorithms for verifying that programs are secure in the t-threshold probing model. The algorithms implement divide-and-conquer strategies and dependency analyses to minimize the set of observations for which information flow checking needs to be performed. In particular, the algorithms first try to find maximal sets $\mathcal{O}_1, \ldots, \mathcal{O}_n$ such that $p_1 \sim_{\mathcal{I}}^{\mathcal{O}} p_2$, and then check whether all sets of size less than t are covered by $\mathcal{O}_1, \ldots, \mathcal{O}_n$. While this second step is an instance of the Hitting Set problem [26], which is known to be NP-hard, our algorithms perform reasonably well in practice, as illustrated in Section 6.

2.7 Related work

We conclude this section with a brief mention of two existing methods for security of masked and multi-party implementations.

Bayrak, Regazzoni, Novo and Ienne [10] develop a 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.

Pettai and Laud [40] prove non-interference with respect to several sets of observations imposed by their adversary model in the context of multi-party computation (MPC). They do so by propagating information regarding linear dependencies on random variables throughout their arithmetic circuits and, using this dependency information, progressively replacing subcircuits with random gates. They do not encounter the same scalability issues we deal with, since they only have to prove non-interference with respect to three different sets of observations (corresponding to the corruption of each party) instead of a combinatorial number. However, they perform their non-interference proofs in the presence of an active adversary that may send invalid data to incoming communication nodes. Their techniques to deal with active adversaries in the MPC model could be adapted in our setting to consider adversaries that may inject limited faults in addition to probing intermediate variables or transitions.

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 in Section 2.5 to a particularly 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 $\sigma_1 \times \ldots \times \sigma_n \rightarrow \tau$, 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 $e = f_1(\ldots, e^1_{i_1-1}, f_2(\ldots f_n(\ldots, e^n_{i_n-1}, x, \ldots) \ldots), \ldots)$, $\forall i\ j,\ x \notin \mathsf{rvar}(e^j_i)$, 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.

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, i.e. $\mathsf{dvar}(\boldsymbol{e}) \subseteq \boldsymbol{x_L} \uplus \boldsymbol{x_H}$. 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 $x \in \boldsymbol{x_L}$.

The proof system for deriving valid judgments is given in Figure 1. The rule (Indep) states that a judgment is valid whenever all the deterministic variables in expressions are public. The rule (Conv) states that one can replace expressions by other expressions that are provably equivalent with respect to the equational theory $\mathcal{E}$. The rule (Opt) states that, whenever the only occurrences of a random variable r in $\boldsymbol{e}$ is 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 $[\![f(e_0, \ldots, e_{i-1}, r, e_{i+1}, \ldots, e_k)]\!] = [\![r]\!]$, since f is i-invertible. 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 2.5, we have shown that the problem of proving that a program is $(\oplus, \mathcal{O})$-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 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), *i.e.* it tries 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 then it 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 $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e})$        ▷ the joint distribution of $\boldsymbol{e}$ is independent from $\boldsymbol{x_H}$
2:     if $\forall x \in \mathsf{dvar}(\boldsymbol{e}).\ x \notin \boldsymbol{x_H}$ then
3:         return Indep
4:     $(\boldsymbol{e'}, e, r) \leftarrow \mathsf{choose}(\{(\boldsymbol{e'}, e, r) \mid e \text{ is invertible in } r \wedge r \in \mathcal{R} \wedge \boldsymbol{e} = \boldsymbol{e'}[e/r]\})$
5:     if $(\boldsymbol{e'}, e, r) \neq \bot$ then
6:         return Opt$(e, r)$ : $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e'})$
7:     return $\bot$
```

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

Our second algorithm (Algorithm 2) is a slight improvement on Algorithm 2 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 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.

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

```
1: function $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}, b)$        ▷ the joint distribution of $\boldsymbol{e}$ is independent from $\boldsymbol{x_H}$
2:     if $\forall x \in \mathsf{dvar}(\boldsymbol{e}).\ x \notin \boldsymbol{x_H}$ then
3:         return Indep
4:     $(\boldsymbol{e'}, e, r) \leftarrow \mathsf{choose}(\{(\boldsymbol{e'}, e, r) \mid e$ is invertible in $r \wedge r \in \mathcal{R} \wedge \boldsymbol{e} = \boldsymbol{e'}[e/r]\})$
5:     if $(\boldsymbol{e'}, e, r) \neq \perp$ then
6:         return Opt$(e, r)$ : $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e'}, b)$
7:     else if $b$ then
8:         $\boldsymbol{e} \leftarrow \mathsf{ring\_simplify}(\boldsymbol{e})$
9:         return Conv : $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}, false)$
10:    return $\perp$
```

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 6 relies on this algorithm.[5]

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. We believe that 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 $\mathcal{O} \in \mathcal{P}_{\leq t}(\mathsf{PVar(p)})$. Simply enumerating all possible observation sets quickly becomes untractable 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 forall subset $\mathcal{O}'$ of $\mathcal{O}$ we have $\mathsf{NI}_{\mathcal{R},\mathcal{O}}(p)$. So checking that forall i, $\mathsf{NI}_{\mathcal{R},\mathcal{O}_i}(p)$ can done in one step by checking $\mathsf{NI}_{\mathcal{R},\cup_i \mathcal{O}_i}(p)$.

The idea is to try 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, 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.

[5] Some of the longer-running experiments reported in Section 6 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.

We describe and implement several algorithms based on the observation that the sequences of derivations constructed to prove the independence judgments in Section 2 allow us 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 $\boldsymbol{x_H}$. In both of the algorithms presented below, we use the sequence of optimistic sampling operations that serves as witness to the independence of X to extend the observation set.

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

Algorithm 3 Rechecking a derivation

```
function $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}, \boldsymbol{h})$ ▷ Check that the derivation represented by $\boldsymbol{h}$ can be applied to $\boldsymbol{e}$
    if $\boldsymbol{h} = \textsc{Indep}$ then
        return $\forall x \in \mathsf{dvar}(\boldsymbol{e}).\ x \notin \boldsymbol{x_H}$
    if $\boldsymbol{h} = \textsc{Opt}(e, r) : \boldsymbol{h'}$ then
        $(\boldsymbol{e}') \leftarrow \mathsf{choose}(\{\boldsymbol{e}' \mid \boldsymbol{e} = \boldsymbol{e'}[e/r]\})$
        if $\boldsymbol{e}' \neq bot$ then
            return $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}', \boldsymbol{h}')$
    if $\boldsymbol{h} = \textsc{Conv} : \boldsymbol{h'}$ then
        $\boldsymbol{e} \leftarrow \mathsf{ring\_simplify}(\boldsymbol{e})$
        return $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}, \boldsymbol{h}')$
```

on $\boldsymbol{e}$. A key observation is that if $\mathsf{NI}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}) = \boldsymbol{h}$ then $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e}, \boldsymbol{h})$. Futhermore, 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 $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{e} \cup \boldsymbol{e}', \boldsymbol{h})$.

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

Algorithm 4 Extending the Observation using a Fixed Derivation

```
function $\mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{x}, \boldsymbol{e}, \boldsymbol{h})$ ▷ find an $\boldsymbol{x} \subseteq \boldsymbol{x'} \subseteq \boldsymbol{e}$ such that $\boldsymbol{h}(\boldsymbol{x'})$ is syntactically independent from $\boldsymbol{x_H}$
    $e \leftarrow \mathsf{choose}(\boldsymbol{e})$
    if $\mathsf{recheck}_{\mathcal{R},\boldsymbol{x_H}}(e, \boldsymbol{h})$ then
        return $\mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}((\boldsymbol{x}, e), \boldsymbol{e} \setminus \{e\}, \boldsymbol{h})$
    else
        return $\mathsf{extend}_{\mathcal{R},\boldsymbol{x_H}}(\boldsymbol{x}, \boldsymbol{e} \setminus \{e\}, \boldsymbol{h})$
```

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, spending 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 6.

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 a $\widehat{X}$ 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 results of recursive calls.

Pairwise Space-Splitting. Our first algorithm (Algorithm 5) uses its initial t-uple 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 $0 < t$, 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 in the traditional way. 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 $x_H$
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 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 $0 < t$, 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 [17] (Algorithm 4) protected at various orders. Table 1 shows the results, where column *# tuples* contains the total number of t-tuples of program points, 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[6] 64 bits 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 6, 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

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

study. In the unlikely event where this occurs in *all* recursive calls, both algorithms degrade into an exhaustive enumeration of all t-tuples.

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 Proving Functional Correctness

In Sections 2 and 4, we discussed how relational program verifications techniques can be used in practice to prove that a program is t-non-interfering. It remains for us to show how the same program verification techniques can be applied to prove the functional equivalence of masked programs with their unprotected version (seen as reference implementations).

Unlike t-non-interference, functional correctness is a compositional property: correctness of say, a masked multiplication algorithm can be used *as is* to prove the correctness of a masked S-Box computation. Although the general problem of taking a masked implementation and *automatically* proving it correct with respect to a functional specification remains hard, it is in fact easy to prove small[7] masked components correct using only EasyCrypt's normalization tactic for algebraic expressions.

Essentially, the non-relational version of the techniques discussed in Section 2.5 can be used to turn a functional correctness objective of the form $\{\boldsymbol{args} = \boldsymbol{x}\}\, p\, \{\mathsf{res} = f(\boldsymbol{x})\}$ for all $\boldsymbol{x}$, where f is a functional specification for program p, into $\{\boldsymbol{args} = \boldsymbol{x}\}\, S\, \{e_{\mathsf{res}} = f(\boldsymbol{x})\}$ for all $\boldsymbol{x}$, where S is a sequence of random samplings and e_{res} an expression describing the output of program p as a function of its input and the random variables sampled in S. Since we are proving functional correctness, the postcondition needs to hold for all possible values of the random variables, and we simply replace S with a universal quantification on all the variables (say $\boldsymbol{r}_S$ it samples, yielding the following verification condition.

$$\forall \boldsymbol{x}\ \boldsymbol{r}_S,\ e_{\mathsf{res}} = f(\boldsymbol{x})$$

We then rely on the equational theory $\mathcal{E}$ to discharge this goal. An example is described in Appendix B.

[7] Around the size of Coron's secure algorithm for computing functions of the form $x \mapsto x \odot g(x)$ with g a linear function [15]. The masking order only has very little influence on this algorithm's complexity.

Once the correctness theorems of such small components are obtained in this way, they can be composed into larger masked functions, whose correctness can be proved compositionally in EasyCrypt's interactive proof engine, with very little program verification expertise. Indeed, in the absence of global state, functional correctness theorems for the underlying procedures can be used to simply replace all calls to the procedure with its functional specification in turn, eventually obtaining a functional correctness theorem for the complex operation.

Note that even these more complex proofs can be performed systematically once individual operations are put in correspondence with a basic operation. As such, a compiler such as those suggested by Ishai, Sahai and Wagner [29] or Duc, Dziembowski and Faust [18], or a code generator for specialized masked implementations could be very easily adapted to produce such proofs along with the masked program, providing an easy way to validate the transformations they perform from a functional correctness point of view, without having to certify them.

6 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 tool's internal representations of programs and expressions, and relying on some of its low-level tactics. 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 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 particular 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.

6.1 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 t-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 two colored rows correspond to examples on which the tool fails to prove t-non-interference.

On Schramm and Paar's table-based implementation of the AES Sbox, supposed to be secure at order 4, our tool finds 98176 third order observations that it cannot prove independent from the secrets. The time expressed is the time needed to cover all triples, the first error is found in 0.221s. These in fact correspond to four families of "bad" 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. $(y_0, \mathsf{Sbox}(X \oplus x_0 \oplus i) \oplus (Y \oplus y_0), \mathsf{Sbox}(X \oplus x_0 \oplus j) \oplus (Y \oplus y_0))$, $\forall i, j \in \mathsf{GF}(2^8),\ i \neq j$

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 [45]	multiplication	13	secure ✓	7	ε
FSE13 [17]	Sbox (4)	63	secure ✓	17	ε
FSE13 [17]	full AES (4)	17,206	secure ✓	3,342	128
Second Order Masking					
RSA06 [47]	Sbox	1,188,111	secure ✓	4,104	1.649
CHES10 [45]	multiplication	435	secure ✓	92	0.001
CHES10 [45]	Sbox	7,140	1^{st}-order flaws (2)	866	0.045
CHES10 [45]	key schedule [17]	23,041,866	secure ✓	771,263	340,745
FSE13 [17]	AES 2 rounds (4)	25,429,146	secure ✓	511,865	1,295
FSE13 [17]	AES 4 rounds (4)	109,571,806	secure ✓	2,317,593	40,169
Third Order Masking					
CHES10 [45]	multiplication	24,804	secure ✓	1,410	0.033
FSE13 [17]	Sbox(4)	4,499,950	secure ✓	33,075	3.894
FSE13 [17]	Sbox(5)	4,499,950	secure ✓	39,613	5.036
Fourth Order Masking					
RSA06 [47]	Sbox	4,874,429,560	3^{rd}-order flaws $(98, 176)$	35,895,437	22,119
CHES10 [45]	multiplication	2,024,785	secure ✓	33,322	1.138
FSE13 [17]	Sbox (4)	$2,277,036,685$	secure ✓	3,343,587	879
Fifth Order Masking					
CHES10 [45]	multiplication	216,071,394	secure ✓	856,147	45

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.

1. The first family corresponds to the attack detailed by Coron, Prouff and Rivain [16]). 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.

2. To recover secrets from an observation of the second kind, the attacked can sum the second and third variables to obtain $X \oplus x_0$, from which it can learn $Y \oplus y_0$ (by combining it with the second variable). Since the adversary also knows y_0, this concludes the attacks.

3. The third family is a variant of the first: the S-box masks can be removed in both cases.

4. Finally, when observing three variables in the fourth family of observations, the knowlege of both x_0 and y_0 unmask 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 [45], that in fact correspond to the two flaws revealed in [17]. 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 [18]) or refreshing all modified variables that are reused later on (as suggested by [42]. Others focused on making use of specialized versions of the multiplication algorithm [17] that allow the masked program to retain its performance whilst gaining in security.

Table 3. Fixing RP-CHES10 [45] at the second order

Reference	S-box	# tuples	Result	Complexity	
				# sets	time
Second Order Masking					
RP-CHES10 [45]	initially proposed	7,140	1^{st}-order flaws (2)	840	0.070s
RP-CHES10 [45]	different refreshMasks	7,875	secure ✓	949	0.164s
RP-CHES10 [45]	more refreshMasks	8,646	secure ✓	902	0.180s
CPRR-FSE13 [17]	use of $x \cdot g(x)$ (Algo 4)	12561	secure ✓	619	0.073
CPRR-FSE13 [17]	use of tables (Algo 5)	12,561	secure ✓	955	0.196

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 variables that appear before it in the program, whereas the dependencies in full AES are sparser. Note that this is reminiscent of limitations inherent to Eldib, Wang and Schaumont's approach [21]. However, although our approach's complexity still depends on the size of the expressions observed by the adversary, this dependency is not inherently exponential, allowing us to consider larger functionalities.

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, which 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.

6.2 Transition-based Model

The value-based leakage model may not always be the best fit to capture the behavior 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 was only secure at the second order in the transition-based leakage model and used 22 local registers, our first improved implementation achieves the third-order security in the transition-based leakage model with only 6 local registers. Trying to provide the best possible security in this model, we also found a third implementation that achieves security at order 4. This last implementation is in fact the original implementation with

Table 4. Multiplication in the transition-based leakage model

Reference	Multiplication	# tuples	Security	Complexity	
				# sets	time
RP-CHES10 [45]	initial scheme for order 4	3,570	order 2	161	0.008s
RP-CHES10 [45]	with some instructions reordering	98,770	order 3	3,488	0.179s
RP-CHES10 [45]	using more registers	2,024,785	order 4	17,319	1.235s

additional registers. Note however, 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 counter-measure 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 counter-measure from operating properly against higher order adversaries.

7 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 levels 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 use 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 the most efficient program p' that is observationally equivalent to $\overline{p}_t$ and equally secure—the latter property being verified using pRHL.

Acknowledgments. We thank François-Xavier Standaert and Vincent Grosso for giving us access to their examples and code generation tools.

References

1. J. Akinyele, G. Barthe, B. Grégoire, B. Schmidt, and P.-Y. Strub. Certified synthesis of efficient batch verifiers. In *27th IEEE Computer Security Foundations Symposium, CSF 2014*. IEEE Computer Society, 2014. To appear.
2. R. J. Anderson and M. G. Kuhn. Low cost attacks on tamper resistant devices. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997.
3. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert. On the cost of lazy engineering for masked software implementations. Cryptology ePrint Archive, Report 2014/413, 2014. `http://eprint.iacr.org/2014/413`.
4. G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, editors, *Principles and Practice of Parallel Programming (PPoPP)*, pages 123–134. ACM, 2013.
5. G. Barthe, M. Daubignard, B. M. Kapron, Y. Lakhnech, and V. Laporte. On the equality of probabilistic terms. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 46–63. Springer-Verlag, 2010.

6. G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.
7. G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Aug. 2011.
8. G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.
9. A. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.
10. A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In G. Bertoni and J.-S. Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 293–310. Springer, Aug. 2013.
11. J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. In H. Handschuh and A. Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 69–83. Springer, Aug. 2004.
12. D. Canright and L. Batina. A very compact "perfectly masked" S-box for AES. In *ACNS*, pages 446–459, 2008.
13. C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain. Higher-order masking schemes for S-boxes. In A. Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 366–384. Springer, Mar. 2012.
14. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Aug. 1999.
15. J.-S. Coron. Higher order masking of look-up tables. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, May 2014.
16. J.-S. Coron, E. Prouff, and M. Rivain. Side channel cryptanalysis of a higher order masking scheme. In P. Paillier and I. Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 28–44. Springer, Sept. 2007.
17. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In S. Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, Mar. 2013.
18. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, May 2014.
19. S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *49th FOCS*, pages 293–302. IEEE Computer Society Press, Oct. 2008.
20. H. Eldib and C. Wang. Synthesis of masking countermeasures against side channel attacks. In A. Biere and R. Bloem, editors, *Computer Aided Verification, CAV 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014.
21. H. Eldib, C. Wang, and P. Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2014.
22. H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont. QMS: evaluating the side-channel resistance of masked software from source code. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 1–6. ACM, 2014.
23. S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In H. Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 135–156. Springer, May 2010.
24. M. Fredrikson and S. Jha. Satisfiability modulo counting: a new approach for analyzing privacy properties. In T. A. Henzinger and D. Miller, editors, *Computer Science Logic – Logic in Computer Science*, page 42, 2014.
25. G. Fumaroli, A. Martinelli, E. Prouff, and M. Rivain. Affine masking against higher-order side channel analysis. In A. Biryukov, G. Gong, and D. R. Stinson, editors, *SAC 2010*, volume 6544 of *LNCS*, pages 262–280. Springer, Aug. 2010.
26. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

27. L. Genelle, E. Prouff, and M. Quisquater. Thwarting higher-order side channel analysis with additive and multiplicative maskings. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 240–255. Springer, Sept. / Oct. 2011.
28. C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.
29. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Aug. 2003.
30. H. Kim, S. Hong, and J. Lim. A fast and provably secure higher-order masking of AES S-box. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 95–107. Springer, Sept. / Oct. 2011.
31. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Aug. 1999.
32. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.
33. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
34. S. Micali and L. Reyzin. Physically observable cryptography (extended abstract). In M. Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 278–296. Springer, Feb. 2004.
35. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 58–75. Springer, Sept. 2012.
36. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Language Design and Implementation (PLDI)*, volume 33, pages 333–344. ACM Press, 1998.
37. E. Oswald and S. Mangard. Template attacks on masking - resistance is futile. In M. Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 243–256. Springer, Feb. 2007.
38. E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen. A side-channel analysis resistant description of the AES S-box. In H. Gilbert and H. Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 413–423. Springer, Feb. 2005.
39. E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater. Improved higher-order side-channel attacks with FPGA experiments. In J. R. Rao and B. Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 309–323. Springer, Aug. / Sept. 2005.
40. M. Pettai and P. Laud. 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`.
41. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.
42. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, May 2013.
43. E. Prouff and T. Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 63–78. Springer, Sept. / Oct. 2011.
44. M. Rivain, E. Dottax, and E. Prouff. Block ciphers implementations provably secure against second order side channel analysis. In K. Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 127–143. Springer, Feb. 2008.
45. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F.-X. Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Aug. 2010.
46. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
47. K. Schramm and C. Paar. Higher order masking of the AES. In D. Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 208–225. Springer, Feb. 2006.
48. F.-X. Standaert, T. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In A. Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 443–461. Springer, Apr. 2009.
49. N. Veyrat-Charvillon, B. Gérard, and F.-X. Standaert. Soft analytical side-channel attacks. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 282–296. Springer, Dec. 2014.

A Initial Transformations on Programs: An Example

To illustrate our algorithms, we consider the simple masked multiplication algorithm defined in [45] 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 [45]

```
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:     $r_{0,1} \overset{\$}{\leftarrow} \mathbb{F}_{256}$
 3:     $r_{0,2} \overset{\$}{\leftarrow} \mathbb{F}_{256}$
 4:     $r_{1,2} \overset{\$}{\leftarrow} \mathbb{F}_{256}$
 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 $[\![c_0, c_1, c_2]\!]$
```

When given a program whose inputs have been annotated as secret or public, we transform it as described at the end of Section 2.5 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;[8] 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.

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.

[8] 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.

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 $[c_0 \oplus c_1 \oplus c_2]$
```

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.)

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. 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.

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 $\oplus r_{1,2} \oplus a_1 \odot b_2 \oplus a_2 \odot b_1$

B Functional Correctness: An Example

We illustrate the automated process to prove functional correctness properties of masked algorithms for basic operations. Here again, we consider function SecMult (Algorithm 7), which can be given the following specification.

$$\forall\, a, b.\ \{a_0 \oplus a_1 \oplus a_2 = a \wedge b_0 \oplus b_1 \oplus b_2 = b\}\ \textsc{SecMult}\ \{\mathsf{res}_0 \oplus \mathsf{res}_1 \oplus \mathsf{res}_2 = a \odot b\} \tag{1}$$

This specification is interpreted as follows.

> For all a and b in $\mathbb{F}_{256}$, whenever SecMult is run on arguments such that $[\![a_0, a_1, a_2]\!]$ is a valid sharing of a and $[\![b_0, b_1, b_2]\!]$ is a valid sharing of b, then all reachable final states are such that $[\![\mathsf{res}_0, \mathsf{res}_1, \mathsf{res}_2]\!]$ (where res represents the function's return value) is a valid sharing of $a \odot b$.

To prove this specification, EasyCrypt's *weakest precondition* calculus computes a first-order logic formula ξ, the *verification condition*, whose validity entails specification (1).

$$\forall\, r_{0,1}, r_{0,2}, r_{1,2}.\ c_0 \oplus c_1 \oplus c_2 = (a_0 \oplus a_1 \oplus a_2) \odot (b_0 \oplus b_1 \oplus b_2) \tag{2}$$

where
$c_0 = a_0 \odot b_0 \oplus r_{0,1} \oplus r_{0,2}$,
$c_1 = a_1 \odot b_1 \oplus r_{0,1} \oplus a_0 \odot b_1 \oplus a_1 \odot b_0 \oplus r_{1,2}$, and
$c_2 = a_2 \odot b_2 \oplus r_{0,2} \oplus a_0 \odot b_2 \oplus a_2 \odot b_0 \oplus r_{1,2} \oplus a_1 \odot b_2 \oplus a_2 \odot b_1$.

This verification condition is then discharged automatically by ring normalization. In theory, the same technique could be applied to prove functional correctness properties of larger masked implementations. However, the size of the expressions involved in the verification condition quickly increases with the size of the program, and the prototype currently reaches its practical limits on larger programs.