`maskVerif`: a formal tool for analyzing software and hardware masked implementations

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

[1] IMDEA Software Institute
`gilles.barthe@imdea.org`
[2] CryptoExperts
`sonia.belaid@cryptoexperts.com`
[3] Rennes Univ
`pierre-alain.fouque@univ-rennes1.fr`
[4] Inria Sophia Antipolis
`benjamin.gregoire@sophia.inria.fr`

Abstract. Masking is a popular countermeasure for protecting both hardware and software implementations against differential power analysis. A main strength of software masking is that its security guarantees can be captured formally through well-established models. The existence of such models, and their relation with probabilistic information flow studied in formal methods, has been instrumental to the emergence of fully automated methods for analyzing masked implementations. In particular, state-of-the-art tools such as `maskVerif` (Barthe *et al.*, EUROCRYPT 2015), have been used successfully for analyzing masked implementations at high orders. In contrast, security models for hardware implementations have remained somewhat less developed, and no prior verification tool has accommodated hardware-specific sources of vulnerabilities such as glitches. Recently, Bloem *et al.* formalize the security of masked hardware implementations against glitches and give a method based on SAT solvers for verifying security automatically. However, their method works for small functionalities and low orders.
In this paper, we extend `maskVerif` tool (Barthe *et al.*, EUROCRYPT 2015) with a unified framework to efficiently and formally verify both software and hardware implementations. In this process, we introduce a simple but expressive intermediate language. Our representation requires that each instruction is instrumented with leakage expressions that may depend on the expressions that arise in the instruction and on previous computation. Despite its simplicity, our intermediate representation covers a broad range of models from the literature; moreover, it is also easily amenable to efficient formal verification. Our results significantly improve over prior work, both in terms of coverage and efficiency. In particular, we demonstrate that our tool is able to analyze examples from (Bloem et al, EUROCRYPT 2018) much faster and at high orders.

Keywords: Glitches, Masking, Formal verification

1 Introduction

Side-channel attacks provide an effective vector to retrieve key material and more generally secret information from cryptographic implementations. Informally, side-channel attacks exploit physical information, such as timing or noise, that can be observed from the execution of said implementations, and carry statistical analysis to retrieve the desired information from these observations. There exist many successful forms of side-channel attacks. For implementations on embedded devices, one of the most devastating family of attacks is differential power analysis (DPA) [24]. Protecting against such attacks is therefore a major theoretical and practical concern, and has been the subject of a long line of research.

The most deployed countermeasure so far is *masking*. Masking uses secret-sharing techniques and rests on the observation that combining $t+1$ data from their noisy leakage was proven to be exponentially hard in t [10, 30]. Given a masking order t, which models the adversary's capabilities, a masking transformation aims to replace every sensitive variable x of the implementation by a set of $t+1$ variables $(x_i)_{0\leq i\leq t}$, called *shares*, such that t of these shares are generated uniformly at random and the last one is computed such that $x_0 \star \dots x_t = x$ for some law $\star$. The main challenge behind masking transformations is to transform computations that were initially performed on inputs x into computations that are performed on $(x_i)_{0\leq i\leq t}$, while very carefully avoiding to compute intermediate results that depend on all the shares of secret inputs. While this goal is reasonably easy to achieve for the case $t = 1$, it is more difficult to achieve for higher values of t. In particular, the complexity of linear functions is multiplied by $t+1$ and the complexity of non-linear functions is often quadratic in t under a masking transformation.

Motivated by the complexity of designing masking transformations, and the desire to reason formally about their correctness, Ishai, Sahai, and Wagner introduced the t-threshold probing model [23]. Under this model, an implementation is secure if and only if any set of t intermediate variables is independent from the secret. While this model makes sense if an attacker can put probes on a circuit at very specific locations, it is mainly justified by the difficulty to combine several variables which jointly depend on the secret. Nevertheless, it does not perfectly fit the reality of embedded devices which may leak noisy functions of all the intermediate variables. The latter is actually captured by another model referred to as *noisy leakage model* that was introduced by Chari et al. [10] and then extended by Prouff and Rivain [30]. This very practical model nevertheless suffers from making security proof very complicated. In order to get use of the benefits from these two models, Duc, Dziembowski, and Faust managed to exhibit a reduction [15] which basically states that an t-threshold probing secure implementation is also secure in the noisy leakage model under a certain level of noise which depends, inter alia, on the masking order t.

The equivalence established in [15] has justified (sometimes a posteriori) an established practice of proving security of implementations in the threshold probing model. Moreover, it has served as a source of motivation for a series of formal methods and automated tools for proving security, or for generating secure masked implementations in the threshold probing model [28, 17, 2, 3, 11, 37]. An interesting by-product of this work, specifically [3] is a stronger notion of security, called strong non-interference, that addresses the long-standing problem of compositional reasoning for masked implementations.

While these models make sense in a software scenario where we assume that an observation reflects the leakage of a single variable, many masking schemes are insecure when executed on hardware devices. The discrepancy between these models and hardware implementations was first analyzed by Mangard, Pop and Gammel [25]. Their work identifies so-called a special kind of transitions, known as *hardware glitches*, as the main origin of the discrepancy. Informally, glitches happen whenever information does not propagate simultaneously within the different wires of a combinatorial logic. Glitches introduce unexpected dependencies between several variables within the same combinatorial logic, and induce additional leakage that can be used to break implementations. Following the seminal work of Mangard, Pop and Gammel [25], several glitches attacks have thus been successfully applied to t-probing secure masking schemes [26, 27, 33, 32]. Many techniques have been developed against glitch attacks. In seminal work, Nikova *et al.* [29, 8] introduce a hardware countermeasure, called Threshold Implementations, a.k.a. TI, in consists in ensuring the three following properties: (i) correctness, the sum of the output shares is equal to the result of applying the function without masking, (ii) incompleteness, each output share shall be computed from at most t input shares, or at least one input share must not be used in the computation of each output share (it guarantees that each output share computation will not leak information on sensitive variables), (iii) uniformity, the output sharing must be uniformly distributed if the input sharing is. A special uniformity technique has been proposed by Daemen in [14]. Finally a generalization of TI, called Consolating Masking Scheme (CMS) [33] by Reparaz *et al.*, and a new one Domain Masking Scheme (DMS) [21] to scale high order masking has been proposed by Gross *et al.* since one of the drawback of TI is the randomness required at the end of each non-linear stage.

In a recent breakthrough, Bloem, Groß, Iusupov, Könighofer, Mangard and Winter [9] address the problem of proving automatically the security of masked implementations in presence of glitches. Their contribution is three-fold. First, they define a new security model, henceforth referred to as the threshold probing model with glitches, which combines the benefits of the threshold probing model with an abstract but accurate modeling of glitches. Second, they propose an automated SMT-based verification method for proving that an implementation is secure in their model. Their verification method, which applies to the first-order and higher-order settings, is based on an estimation of the Fourier coefficients for all gates of the circuit. Third, they apply their method on a representative set of examples, including the S-Boxes of AES, Keccak and FIDES.

Contributions

The first major contribution of the paper is an alternative method for proving security of masked implementations in the threshold probing model with glitches. The initial observation is that specialized proof systems for probabilistic non-interference [2, 3], suffice for verifying the examples presented in [9].

The second main contribution of the paper is a unified framework for verifying masked implementations. The framework relies on a simple but expressive intermediate representation, called `MaskIR`, in which each instruction is explicitly annotated by its leakage, which is itself modelled as an expression in an extended language. The intermediate representation naturally captures several models from the literature, including the threshold probing model, the bounded moment model, and the threshold probing model with glitches. Moreover, it can be used to establish comparisons between the different models. In order to support formal verification of implementations written in `MaskIR`, we present a new implementation of `maskVerif`, with new functionalities and several major efficiency improvements.

We then use the new implementation to verify security of masked implementations in presence of glitches, and also to revisit other examples in different models. Overall, our results show significant improvements over prior work in terms of coverage and efficiency. For instance, the first-order hardware implementation of the AES s-box provided by Grosso et al. [21] is proven to be secure with presence of glitches within less than three seconds.

Interestingly, verifying an hardware implementation with the t-probing model with glitches is often faster than verifying the same implementation without glitches. This can be explained by a very advantageous property of `maskVerif` which only verifies relevant tuples (possibly of larger size than the verification order) that are not included in any other ones. For instance, it is not necessary to take into account the observation of a single intermediate variable when this same variable can be observed in the glitchy model with several other ones. Consequently, the sets to analyze are generally much bigger but they are mostly much less numerous, which suffices to make the verification faster. Concrete experiments are provided in Section 6.

General remarks We conclude this introduction with some general remarks with respect to related work.

From a high-level perspective, our work reinforces the observation that the same masking algorithm can be used for software and hardware implementations. This observation is for instance made formal by Gross and Mangard [20]. Our work goes one step further by illustrating that the same verification techniques can be used for hardware and software.

Verification approaches generally make trade-offs between efficiency and precision. Efficient approaches are often desirable, specially in the context of masking, because of the combinatorial complexity of proving that an implementation is secure. Our work demonstrates, through concrete examples, that our more efficient method can be used to verify examples beyond the reach of [9]. However, more precise approaches remain important, when verification with more efficient methods fail. In particular, more precise approaches may be able to detect whether verification fails because of the cruder approximations made by efficient approaches, or because the implementation is not secure. In light of the above discussion, a desirable objective would be to integrate all available methods, including a non-approximate version of the algorithms from [9], in a single tool, and to make them cooperate in a way that achieves the best trade-offs. We contend that our work makes a preliminary step in this direction, but additional work remains necessary to realize this goal.

Finally, formal verification complements, rather than supplants, existing approaches to perform an empirical assessment of the leakage, using on statistical tests or concrete attacks. While practical, empirical approaches are not designed to guarantee the absence of attacks on a different device or when the adversary is given more leakage traces. On the other hand, formal verification necessarily relies on models, which must be validated empirically.

2 Leakage Models and Existing Verification Approaches

We start with an informal description of the problem and of the main challenges to verify masked implementations. We then proceed with a brief and informal explanation of some of the most important models and security notions. Finally, we conclude with a critical review of prior work on formal verification, providing in each case a brief account of the methods used and of their limitations.

2.1 Problem statement

For the purpose of this section, we place ourselves in an informal setting. We consider probabilistic implementations that operate on inputs, perform intermediate computations, and return outputs. We suppose that implementations come with a set of secrets, typically computed as a deterministic function of the inputs.

The definition of security depends on three notions: the execution model, the leakage model, and the adversary model. Execution models capture the operational behavior (a.k.a. semantics) of programs.

Program behavior is defined compositionally: one defines the behavior of atomic computations in isolation, and then one defines the behavior of program from the behavior of its sub-components. Note that execution models do not capture leakage.

Leakage models define how much information is leaked by each atomic computation. Models differ in two main dimensions, namely origin and nature of leakage:

- *origin*: in simpler models, leakage only depends on the values manipulated by the current computation. However, other models allow leakage to depend on the results of previous sub-computations. Obviously, leakage never depends on future sub-computations. Thus, in contrast to program semantics, program leakage needs not be fully compositional, in the sense that it is not always possible to define leakage of atomic computations in isolation;
- *nature*: in simpler models, atomic computations leak exact values, e.g. their output. In other models, atomic computations leak noisy values, e.g. obtained by adding Gaussian noise to its output, or more generally arbitrary functions of values. These functions may be fixed, or arbitrarily chosen.

Adversary models specify how much information an adversary can gain from an execution of a complete implementation. In some models, the adversary obtains the joint distribution of all atomic leakages. In most models, however, this is not the case. Instead, it is common to assume that adversaries can only observe a maximal number t of atomic leakages, called the order of an adversary. Given a (possibly adversarially chosen) set O of atomic leakages of size $\leq t$, we define the leakage of an implementation w.r.t. O to be the joint distribution of atomic leakages for all elements of O. Throughout the next sections, we let $\mathcal{L}_O(x)$ denote the leakage obtained by an adversary which observes for an observation set O of his choice the execution of the implementation on input x.

Given execution, leakage and adversary models, one can now define the notion of secure implementation. Informally, an implementation is secure if for all possible choices of the adversary, leakage does not reveal any information about secrets. Prior work captures this intuition through two distinctive flavours of security definitions, using notions of "small" and "equivalence" (we make the definition of the latter more precise in Section 2.2):

- in the simulation-based paradigm, one proves that for every set of observations, leakage can be computed from a "small" subset on inputs. More precisely, one exhibits a simulator that takes as input a "small" subset of inputs, and computes the leakage. One then argues that "small" subsets of inputs are (in general uniformly distributed and always) independent from the secret, and concludes that the set of observations reveals nothing to the adversary.
- in the information flow paradigm, one proves that for every set of observations, there exists a "small" subset of inputs, such that leakage is equal for every two runs that coindide on this "small" subset of inputs. As in the previous case, One then argues that "small" subsets of inputs are (in general uniformly distributed and always) independent from the secret, and conclude similarly.

The simulation-based paradigm is familiar in cryptography. However, it involves an existential quantification over simulators. In contrast, the information flow paradigm is familiar in programming languages and formal verification, and only involves universal quantification (over pairs of related inputs). Fortunately, it is often possible to prove equivalence between simulation-based and information flow-based definitions.

Still, verification of masked implementations is a challenge for two main reasons:

- combinatorial explosion. The number of adversarial choices of observation sets grows exponentially with the order t and the size of the implementation [2, 11, 9].
- non-compositionality. The sequential composition of two secure implementations is not always secure [34, 13, 31, 3].

As a consequence, manuel verification of masked implementations at higher orders is at best error-prone and more generally unrealistic. This has been a main source of motivation for developing automated verification tools, and for proposing new, compositional, notions of security. In this work, we propose a unified framework and algorithms for dealing with combinatorial explosions. Moreover, our framework is compatible with recent proposals to achieve compositional verification.

2.2 Security notions

In order to explain the capabilities of the different verification tools, and set the stage for further generalizations, we recall three (still informal) security notions from the literature. We start from the weaker definition, called threshold probing security, and then present an intermediate definition, called non-interference in the literature, and conclude with the strongest notion, called strong non-interference in the literature.

For the clarity of exposition, we consider the case of programs with two inputs. However, all definitions extend without any difficulty to programs with an arbitrary number of inputs. Concretely, we let the inputs be $\boldsymbol{x} = (x_1, \ldots, x_{t+1})$ and $\boldsymbol{x'} = (x'_1, \ldots, x'_{t+1})$, and assume that the secrets that should not be leaked by computation are $s = x_1 + \ldots + x_{t+1}$ and $s' = x'_1 + \ldots + x'_{t+1}$. We also define μ_s as the uniform distribution over all tuples of inputs $(x_1, \ldots, x_{t+1})$ such that $s = x_1 + \ldots + x_{t+1}$. Then, $\mu_{s'}$ is defined similarly.

We shall also need the following definition. Let $I \subseteq \{1, \ldots, t+1\}$. We say that two tuples $\boldsymbol{v} = (v_1, \ldots, v_{t+1})$ and $\boldsymbol{v'} = (v'_1, \ldots, v'_{t+1})$ are I-equivalent, written $\boldsymbol{v} \simeq_I \boldsymbol{v'}$, iff $v_i = v'_i$ for every $i \in I$. We also let $\boldsymbol{v}_I$ denote the subvector containing only indices from I.

Threshold probing security. The first notion is *threshold probing security*, which can be understood informally as a notion of non-interference under uniform inputs. We say that an implementation is t-threshold probing secure or t-non-interfering under uniform inputs, iff for every (s, s') and (u, u'), and every observation set O such that $|O| \leq t$,

$$\mathcal{L}_O(\mu_s, \mu_{s'}) = \mathcal{L}_O(\mu_u, \mu_{u'})$$

Non-interference. We say that an implementation is t-non-interfering, written t-NI, iff for every observation set O such that $|O| \leq t$, there exists two sets I and I' such that $|I|, |I'| \leq t$ and for every pair of inputs $(\boldsymbol{x}, \boldsymbol{x'})$ and $(\boldsymbol{y}, \boldsymbol{y'})$,

$$\boldsymbol{x} \simeq_I \boldsymbol{y} \wedge \boldsymbol{x'} \simeq_{I'} \boldsymbol{y'} \Longrightarrow \mathcal{L}_O(\boldsymbol{x}, \boldsymbol{x'}) = \mathcal{L}_O(\boldsymbol{y}, \boldsymbol{y'}).$$

An interesting observation, established in [6], is that it is equivalent to require $|I|, |I'| \leq t$ and $|I|, |I'| \leq |O|$.

This is equivalent to requiring for every set of observations O the existence of two sets I and I' with $|I|, |I'| \leq t$ and a simulator S that takes as inputs $\boldsymbol{x}_I$ and $\boldsymbol{x'}_{I'}$ such that

$$\mathcal{L}_O(\boldsymbol{x}, \boldsymbol{x'}) = S(\boldsymbol{x}_I, \boldsymbol{x'}_{I'}).$$

Strong non-interference. The final notion is *strong non-interference*, and is used for compositional reasoning. Strong non-interference distinguishes between internal and output observations. For every observation set O, we let $\| O \|$ denote the size of its subset of internal observations. Then, we say that an implementation is t-strong non-interfering, written SNI, iff for every observation set O such that $|O| \leq t$, there exists two sets I and I', such that $|I|, |I'| \leq \| O \|$ and for every pair of inputs $(\boldsymbol{x}, \boldsymbol{x'})$ and $(\boldsymbol{y}, \boldsymbol{y'})$,

$$\boldsymbol{x} \simeq_I \boldsymbol{y} \wedge \boldsymbol{x'} \simeq_{I'} \boldsymbol{y'} \Longrightarrow \mathcal{L}_O(\boldsymbol{x}, \boldsymbol{x'}) = \mathcal{L}_O(\boldsymbol{y}, \boldsymbol{y'}).$$

As for non-interference, strong non-interference admits an equivalent but simulation-based definition of security.

Obviously, it can be verified that strong non-interference implies non-interference which itself implies threshold probing security. More interestingly, the different definitions serve different purposes and have complementary strengths:

- Threshold probing model implies security in the noisy leakage model [15];
- Non-interference (NI) requires showing that every set of at most t intermediate variables can be perfectly simulated with at most t shares of each input. This is generally much easier to demonstrate than threshold probing security which requires that each such set must be statistically independent from the secret. Intuitively, it is easier to determine the dependencies between a set of intermediate variables and some input shares than to reason on distributions. In addition, NI is a simpler definition for reasoning about composition of NI gadgets as observations can be propagated through the circuit (i.e., simulation with some input shares become observations on output variables) to reason on the global security. Non-interference is used as a baseline definition in almost all works on formal verification.
- While the NI property is a convenient target for verification, it cannot be used on its own for analyzing the security of complete implementations. Therefore, Barthe et al. [3] introduce SNI which characterizes gadgets for with output observations are somehow independent from input shares. The SNI property supports compositional reasoning and can be used to justify the security of complete implementations.

2.3 Leakage models

In this paragraph, we review five leakage models from the literature.

Threshold probing model. This model was introduced by Ishai, Sahai, and Wagner [23]. In this model, the adversary chooses a set of intermediate variables of his choice; the requirement is that the size of the set is bounded by some natural number t, called the order.

Leakage is then defined as the joint distribution of the intermediate values. A circuit is said to be *t-private* if and only if, for all adversarial choices of sets X of intermediate variables (subject to the cardinality constraint $|X| \leq t$), the leakage, modelled as a joint distribution, does not reveal any information about the secret.

The conceptual simplicity of the threshold probing model makes it particularly convenient for reasoning formally about the security of masked implementations. It is therefore not suprising that a large majority of prior works, in particular those focused on masking of software implementations, is based on the threshold probing model. Moreover, the threshold probing model has been a main target for formal verification, as detailed in the next subsection.

Noisy leakage model. Another well-known model to reason on the security of masked implementations is the *noisy leakage model*, introduced in 1999 by Chari et al. [10] and later extended by Prouff and Rivain [30]. The noisy leakage model assumes that the attacker has access to noisy functions of all the intermediate variables of the implementation. This provides a realistic model for side-channel attacks on embedded devices.

In 2014, Duc, Dziembowski, and Faust [15] establish a reduction between the probing model and the noisy leakage model. In a nutshell, they prove that a t-probing secure implementation is also secure in the noisy leakage model for a certain level of noise. Beyond its foundational interest, their result is practically very important, since it makes it possible to prove security directly in the t-probing model, and to derive practical guarantees. Further work by Dziembowski, Faust and Skórski [16] improves the equivalence by deriving tighter bounds.

Proving security in the noisy leakage model and its variants involves complex calculations. For this reason, the noisy leakage model has not been used directly in formal verification. Of course, it is possible to rely on the equivalence with the threshold probing model, and to carry the verification in the latter.

Bounded moment model. The *bounded moment model* was recently introduced by Barthe, Dupressoir, Faust, Grégoire, Standaert and Strub [4]. The bounded moment model reasons about parallel implementations, and thus provides an intermediate ground between the threshold probing model and hardware implementations. Formally, the bounded moment model reasons about programs with parallel assignments. The definition of leakage for such parallel assignments is based on their mixed moments. Barthe et al [4] prove that in order for a parallel implementation is secure at order t in the bounded moment model, it suffices that a serialization of the implementation is secure at order t in the threshold probing model.

Proving security in the bounded moment model requires reasoning about expectations. For this reason, the bounded moment model has also not been used directly in formal verification. As for the noisy leakage model, it is always possible to rely on the equivalence with the threshold probing model, and to carry the verification in the latter.

Threshold probing model with transitions. So far, the described models generally assume that intermediate variables of the targeted implementation may leak at the time they are manipulated in the cryptographic algorithms. Nevertheless, it has been observed in the literature [1, 12] that software devices could also leak on pairs of variables when they are consecutively stored in the same register. By opposition to the leakage of single intermediate variables as considered in the original threshold probing model, this extended notion is generally referred to as the *transition-based leakage*. When such a leakage is considered, the t-probing model can be slightly tweaked so that the adversary is allowed to perform t observations such that each observation may capture two variables, namely the targeted variable and the variable that was previously stored in the same register. In that case, the adversary can get up to $2t$ variables to perform its attack, which make the registers allocation critical to avoid higher-order attacks. In summary, the threshold probing model with transitions is a minor variant of the threshold probing model, and as such lies within the scope of formal verification.

Threshold probing model with glitches. Even when they are secure in the threshold probing model, hardware implementations can still be vulnerable to glitches attacks. As a consequence, recent works introduce an extension of the threshold probing model to reason on hardware security. As part as a more generic hardware leakage model, Faust et al. [19] consider an extension of the authorized set of probes in the threshold probing model so that putting one probe on a gadget reveals to the attacker the whole set of this gadget's inputs (assuming the gadget is performed within the same combinatorial logic). Their model rests on two observations: first, glitches completely break locality of leakage, in the sense that computations may leak values that depend on prior computations; this is similar to the model with transitions, except that non-locality arises at a significantly higher scale. Second, glitches may yield unexpected computations. Their model renders these two observations concrete by letting adversaries learn, under provisos that reflect physical constraints imposed by hardware, all the inputs of a combinatorial set at the cost of a single observation.

Similarly, in a recent paper, Bloem, Groß, Iusupov, Könighofer, Mangard and Winter [9] introduce an extension of the threshold probing model, henceforth called, the threshold probing model with glitches. Contrary to Faust et al.'s model which provides the attacker the whole set of inputs for an observation within a combinatorial logic set, Bloem et al's model let adversaries select, within similar hardware constraints, modified implementations on which the observations will be made. Crucially, their definition of security in presence of glitches states that a program P is secure at order t in the threshold probing model with glitches iff a program P' built from P (along the lines discussed above) is also secure at order t in the threshold probing model.

These two models are actually equivalent as soon as authorizing observations on any function of a combinatorial set's inputs is equivalent to authorizing observations on all its inputs. In the rest of this paper, we will refer to these models as the threshold probing model with glitches.

2.4 Formal Verification Tools

In this section, we review the state-of-the-art for the formal verification of masked implementations.

Verification of software-based implementations. Moss, Oswald, Page and Turnstall [28] were the first to consider the use of automated methods to build or verify masked implementations. Specifically, they propose and implement a type-based masking compiler that track which variables are masked by random values and iteratively modifies an unprotected program until all secrets are masked. This strategy suffices to ensure security against first-order differential power analysis, and works well on many examples.

While type-based verification is generally efficient and scalable, it is also often overly conservative, i.e. it rejects secure programs. Logic-based verification often strikes interesting trade-offs between efficiency and expressiveness. This possibility was first explored in the context of masked implementations by Bayrak, Regazzoni, Novo and Ienne [5]. Concretely, they propose a SMT-based method for analyzing the security of masked implementations against first-order differential power analysis. In contrast to the type system of [28], which targets proving a stronger property of programs, their method directly targets proving statistical independence between secrets and leakage. Their approach is limited to first-order masking but was extended to higher orders by Eldib, Wang and Schaumont [17]. Their approach is based on a logical characterization of security, akin to non-interference (NI), and is based on model counting. Unfortunately, model counting incurs an exponential blow-up in the security order, and becomes infeasible even for relatively small orders. Eldib, Wang and Schaumont circumvent the issue by developing sophisticated (and somewhat unintuitive) methods for incremental verification. Although such methods help, the scope of application of their methods remains limited. Recently, Zhang, Gao, Song and Wang [37] show how abstraction-refinement techniques provide significant improvement in terms of precision and scalability. They implement their technique in a tool, called SCInfer, that alternates between fast and moderately precise approaches (partly inspired from [2], which we describe below) and computationally expensive but precise approaches. Their tool delivers practical results for the benchmarks taken from [17].

Independently, Barthe, Belaïd, Dupressoir, Fouque, Grégoire and Strub [2] propose a different approach for proving security in the theshold probing model. Their approach establishes and leverages a tight connection between the security of masked implementations and probabilistic non-interference, for which they propose efficient verification methods. Specifically, they show how a relational program logic previously used for mechanizing proofs of provable security can be specialized into an efficient procedure for proving probabilistic non-interference, and develop techniques that overcome the combinatorial explosion of observation sets for high orders. Informally, the main idea of their algorithm is to carefully select sets of t or more intermediate variables and to repeatedly apply optimistic sampling on the tuple of expressions that represent the results of these intermediate variables until they do not depend on the secret. The concrete outcome of their work is the `maskVerif` framework, which achieves practicality at reasonably high orders. For instance, they report using `maskVerif` to automatically and formally verify the probing security of the original Ishai-Sahai-Wagner multiplication [23] at order 5. A tweaked version of `maskVerif` also offers the possibility to verify the security of higher-order implementations in the transition-based model.

A follow-up work by the same authors [3] addresses the problem of compositional reasoning by introducing the notion of strong non-interference (SNI) discussed in the previous paragraph, and adapts `maskVerif` to check SNI. The adaptation achieves similar coverage as the original tool, i.e. it achieves practicality at reasonably high-orders. In addition, [3] proposes an information flow type system with cardinality constraints, which forms the basis of a compiler, called `maskComp`. This compiler transforms an unprotected implementation into an implementation that is protected at any desired order—the order is passed as an argument. Somewhat similar to the masking compiler of [28], `maskComp` uses typing information to control and to minimize the insertion of mask refreshing gadgets.

More recently, Coron [11] presents an alternative tool, called `checkMasks`. `checkMasks` achieves similar functionalities as `maskVerif`, but exploits a more extensive set of transformations for operating on tuples of expressions. This is useful to achieve better verification times on selected examples.

Verification of hardware-based implementations. The second major breakthrough presented in Bloem, Groß, Iusupov, Könighofer, Mangard and Winter [9] is a formal technique for proving security of implementations in the threshold probing model with glitches. Their method is based on Xiao-Massey lemma, which provides a necessary and sufficient condition for a boolean function to be statistally independent from a subset of its variables. Informally, the lemma states that a boolean function f is statistically independent of a set of variables X if and only if the so-called Fourier coefficients of every non-empty subset of X is null. However, since the computation of Fourier coefficients is computationally expensive, they use instead an approximation method, whose correctness is established in their paper. By encoding their approximation in logical form, they are able to instantiate their approach using SAT-based solvers. Their tool is able to verify implementations of S-Boxes of AES, Keccak and FIDES. However, the cost of the verification is significant.

3 Framework

This section describes our new `MaskIR` intermediate representation to describe software and hardware implementations, and demonstrates its use to represent different leakage models. We also prove general results for transferring security proofs from one model to another, and for optimizing security proofs.

Furthermore, we recall the formal verification algorithms used in the original `maskVerif` and describe their adaptation to `MaskIR` and some extensions, in particular for verifying non-interference under uniform inputs.

3.1 Rationale

The basic idea between `MaskIR` is two-fold: on the one hand, existing leakage models artificially enforce restrictions on security notions. For instance, our definition of input equivalence enforces that every element in the partition of inputs has size $t + 1$, whereas it would suffice that every element in the partition contains at least $t + 1$ elements. While such coincidences can be important for some specific proof techniques, they are never exploited in the approach followed by `maskVerif`.

On the other hand, formal verification is only possible if leakage is made explicit. A common approach to model the non-functional behavior of programs, of which leakage is an instance, is to use ghost code. In our case, ghost code is added at each program point to model its leakage. In principle, one could execute the program, together with its ghost annotations, to compute leakage for chosen sets of inputs. However, we only use ghost code for verification purposes.

Finally, `MaskIR` also provides a convenient framework to compare and establish relations between different leakage models, in the spirit of [4], which proves that security in the threshold probing model entails security in the bounded moment model. We use this ability of `MaskIR` to prove equivalence between the notion of security from [9], in which the leakage functions are adversarially controlled, and a simpler notion of security. In a similar way, `MaskIR` can be used to justify optimizations of the verification algorithm. In particular, we introduce a notion of covering set of observation sets, and prove that the absence of leakage for all observation sets in the covering set suffices to ensure security.

3.2 Programming language and leakage model

The syntax of programs is shown in Figure 1. Programs are modelled as sequences of deterministic and probabilistic assignments. Leakage is modelled explicitly by tagging instructions with an expression ℓ that defines its leakage. More formally, deterministic assignments are of the form $x \leftarrow e \mid \ell$, where $x \in \mathcal{X}$ is a deterministic variable, $e \in \mathcal{E}$ is a program expression, and $\ell \in \mathcal{L}$ is a leakage expression. Probabilistic assignments are of the form $r \leftarrow \mu \mid \ell$, where $r \in \mathcal{R}$ is a probabilistic variable, μ is a distribution expression, and $\ell \in \mathcal{L}$ is a leakage expression. The set $\mathcal{E}$ and $\mathcal{L}$ are defined inductively from variables and operators. In general, leakage expressions may use a richer set of operators than program expressions. In particular, leakage expressions often represent tuples of values, although tuples may not be supported in the core language. Moreover, operators used in leakage expressions may be probabilistic, for instance to return noised values. The operational semantics of `MaskIR` programs produces for each program point a value, and a leakage, both of which may be probabilistic. It is then direct to define the leakage of an observation set, which we model as a set of program points. As before, we write $\mathcal{L}_O(x)$ for the leakage gained by an adversary observing at program points in O of his choice the execution of the implementation under consideration on input x. We note that without loss of generality, we can assume that programs are written in single static assignment (SSA) form, i.e. every variable x or r appears at most once on the left-hand side of an assignment. However, putting a program in SSA form induces a loss of information that is relevant for some models, e.g. when transitions are involved. In this case, we assume given a predecessor function on variables.

Expressions

$$\begin{aligned} e &::= x \mid r \mid f(e_1, \dots, e_n) \\ \ell &::= x \mid r \mid g(\ell_1, \dots, \ell_n) \mid \langle \ell_1, \dots, \ell_n \rangle \end{aligned}$$

where x ranges over variables and r ranges over probabilistic variables, f ranges over deterministic operators from a set $\mathcal{F}$, g ranges over deterministic and probabilistic operators from a set $\mathcal{G}$ (in general $\mathcal{F} \subseteq \mathcal{G}$) and $\langle \dots \rangle$ is syntax for tuples.

Statements

$$\begin{array}{rll} s ::= & x \leftarrow e \mid \ell & \text{deterministic assignment} \\ \mid & r \leftarrow \mu \mid \ell & \text{probabilistic assignment} \\ \mid & s; s & \text{sequential composition} \\ \mid & \mathsf{return}\ e & \text{return expression} \end{array}$$

Fig. 1. Syntax of `MaskIR`

Example 1. We briefly indicate how different leakage models can be encoded in `MaskIR`. The case of glitches is discussed in the next paragraph.

– threshold probing model: we set $\ell = x$ for a deterministic assignment $x \leftarrow e \mid \ell$, and $\ell = r$ for a probabilistic assignment $r \leftarrow \mu \mid \ell$;
– threshold probing model with transitions: in this model, it is assumed that the consecutive storage of two variables in the same register may leak both values during the second assignment. In that case, the leakage ℓ of the second assignment is thus defined as a pair of values corresponding to the successively stored variables. More concretely, we assume that programs are written in SSA form and we assume given a predecessor function on variables. We then set $\ell = \langle x, y \rangle$ for a deterministic assignment $x \leftarrow e \mid \ell$, where y is the predecessor of x;
– noisy leakage model: for each deterministic assignment (resp. probabilistic) assignment, ℓ is expressed as the sum of x (resp. r) and a (generally Gaussian) noise;
– bounded moment model: this model features parallel assignments of the form $(x_1, \dots, x_n) \leftarrow (e_1, \dots, e_n)$. Without loss of generality we can assume that programs are written in SSA form so $x_1, \dots, x_n$ do not occur in $e_1, \dots, e_n$. One can then translate the parallel assignment above into a sequence of assignments

$$x_1 \leftarrow e_1 \mid \epsilon; \dots; x_{n-1} \leftarrow e_{n-1} \mid \epsilon; x_n \leftarrow e_n \mid \ell$$

where ℓ is a vector that contains all the mixed moments $\ell^{(o_1,\dots,o_n)}$ at orders $t_1, t_2, \dots, t_n$ such that $\sum_{1\leq i\leq n} m_o \leq t_o$, and

$$\ell^{(o_1,\dots,o_n)} = \mathsf{E}(x_1^{o_1} \times x_2^{o_2} \times \dots \times x_n^{o_n})$$

3.3 Modeling glitches

In order to handle verification of hardware masked implementations, we rely on the threshold probing model with glitches introduced in two different ways by Faust et al. [19] and by Bloem at al. in [9]. Basically, we consider an hardware implementation as an oriented graph of vertices (i.e., operation gates) and edges (i.e., variables) organized with so-called *combinatorial logic sets* separated by registers. Combinatorial logic sets are sub-graphs that gather operations (vertices) on variables which aims to compute a same output which is to be stored in a register. In hardware, storing a variable in a register creates a synchronization point which stops the propagation of glitches. In the threshold probing model with glitches we rely on, the adversary is thus allowed to make at most t observations on wires, each one resulting in the whole set of inputs that are manipulated so far within the corresponding combinatorial logic set.

Hardware leakage with glitches can easily be described in `MaskIR` programming language. Basically, for each deterministic assignment ℓ is set to a tuple formed by the current x and all the variables involved in the computation of e which belongs to the same combinatorial logic set. For probabilistic assignments, ℓ is simply set to r.

We illustrate the threshold probing model with glitches on a concrete example with the hardware first-order implementation of the DOM AND provided by Grosso, Mangard, and Korak [21] and displayed as a graph in Figure 2. It takes as inputs secrets a and b that are respectively additively split into shares $a.[0]$, $a.[1]$, and $b.[0]$, $b.[1]$. Gates designed by `FF` represent registers, and variable r is a uniform random variable. In the threshold probing model (i.e., without glitches), leakage for each wire is illustrated on Figure 2. Note that leakage at the output of the registers (`FF`) is the same as on its input. In the threshold probing model *with glitches*, leakage for each wire is displayed on Figure 3.

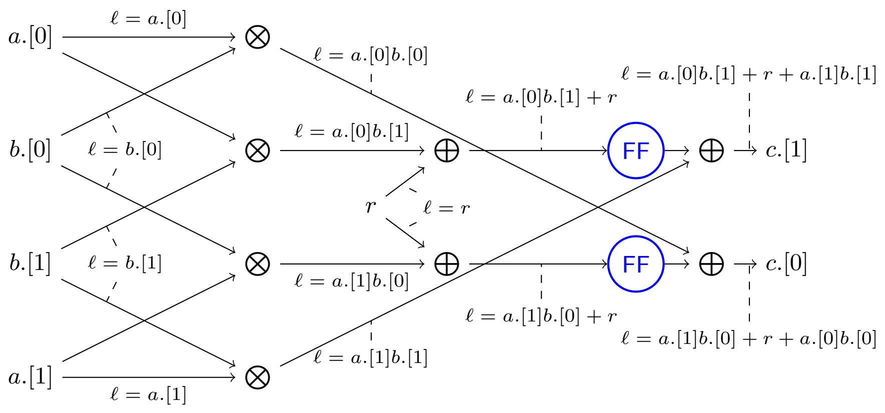

Fig. 2. Graph representation of procedure DOM AND where leakage is assigned according to the threshold probing model (software)

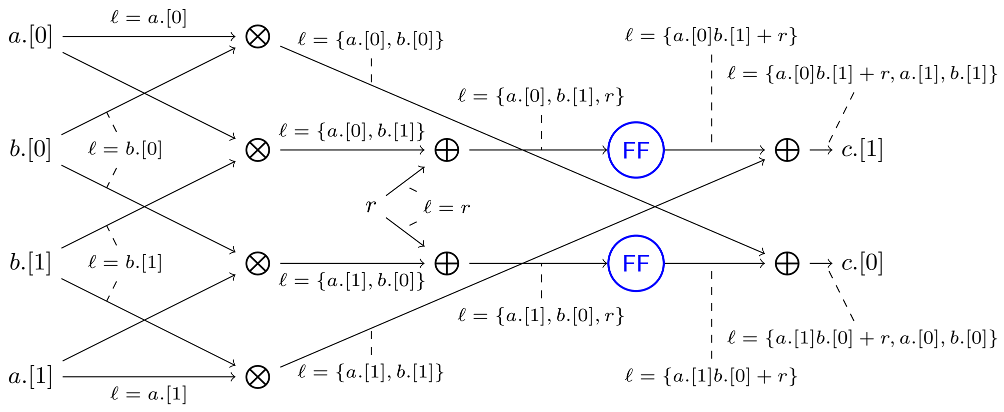
$a.[0]$
$b.[0]$
$b.[1]$
$a.[1]$
$\ell = a.[0]$
$\ell = b.[0]$
$\ell = b.[1]$
$\ell = a.[1]$
$\ell = \{a.[0], b.[0]\}$
$\ell = \{a.[0], b.[1]\}$
$\ell = \{a.[1], b.[0]\}$
$\ell = \{a.[1], b.[1]\}$
r
$\ell = r$
$\ell = \{a.[0], b.[1], r\}$
$\ell = \{a.[1], b.[0], r\}$
FF
FF
$\ell = \{a.[0]b.[1] + r\}$
$\ell = \{a.[1]b.[0] + r\}$
$\ell = \{a.[0]b.[1] + r, a.[1], b.[1]\}$
$\ell = \{a.[1]b.[0] + r, a.[0], b.[0]\}$
$c.[1]$
$c.[0]$

Fig. 3. Graph representation of procedure DOM AND where leakage is assigned according to the threshold probing model with glitches (hardware)

The threshold probing security with glitches can be easily extended to define (S)NI security with glitches as well. Basically, as in the threshold probing security model with glitches, each observation is replaced by the set of intermediate variables that are previously manipulated in the same combinatorial logic set since the previous synchronization point (register). The two (informal) security definitions directly follow.

Definition 1 (NI with glitches). *An implementation is t-NI with glitches iff any set of at most t observations can be perfectly simulated with at most t shares of each input when each observation is replaced by the set of intermediate variables that are previously manipulated in the same combinatorial logic set.*

For SNI security notion, when outputs are not stored in register, then observations on the output are also replaced by the set of intermediate variables that are involved in the current computation since the last register.

Definition 2 (SNI with glitches). *An implementation is t-SNI with glitches iff any set of at most t observations whose t_1 on internal variables and t_2 on output variables can be perfectly simulated with at most t_1 shares of each input when each observation is replaced by the set of intermediate variables that are previously manipulated in the same combinatorial logic set.*

3.4 Comparing models

Thus far, we have shown how `MaskIR` provides a unifying framework for modelling leakage. We now prove that it can also be used for comparing models. To achieve this goal, it is first necessary to define general notions of security that subsume the notions to be compared. This can be done at little expense. First, see that without loss of generality, can assume that each leakage expression corresponds to one observation from the adversary, since expressions can be split into smaller ones to achieve this effect. Then, assume as before that inputs are split into shares; however, we do not require that the number of shares is related to the order t against which the security analysis will be performed. Moreover, we assume given a norm function that maps every observation set O a natural number $\| O \|$, called its norm. Then we say that an implementation verifies general non-interference at order t (and w.r.t. $\| \cdot \|$) iff for every observation set O such that $|O| \leq t$, there exists two sets I and I', such that $|I|, |I'| \leq \| O \|$ and for every pair of inputs $(\boldsymbol{x}, \boldsymbol{x}')$ and $(\boldsymbol{y}, \boldsymbol{y}')$,

$$\boldsymbol{x} \simeq_I \boldsymbol{y} \wedge \boldsymbol{x}' \simeq_{I'} \boldsymbol{y}' \Longrightarrow \mathcal{L}_O(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{L}_O(\boldsymbol{y}, \boldsymbol{y}').$$

Note that this definition can still be generalized in multiple dimensions. However, it suffices for recovering prior definitions and for our purposes.

With this unified definition, `MaskIR` provides a convenient formalism to compare models. In this paragraph, we provide a set of rules for increasing leakage in an implementation. By repeatedly applying these rules, one can reduce security in one model to security in another model.

Definition 3. *The leakage amplification relation $P \rightsquigarrow P'$ is the smallest reflexive transitive relation closed under the following rules:*

- *simplification: this is a local rule which allows to replace a leakage expression by its components:*

$$x \leftarrow e \mid f(\ell_1, \dots, \ell_n) \rightsquigarrow x \leftarrow e \mid \langle \ell_1, \dots, \ell_n \rangle$$

- *extension: this is a local rule which allows to add a leakage expression:*

$$x \leftarrow e \mid \langle \ell_1, \dots, \ell_n \rangle \rightsquigarrow x \leftarrow e \mid \langle \ell_1, \dots, \ell_{n+n'} \rangle$$

- *permutation: this is an administrative local rule which is used to capture the fact that the order of leakage expressions is irrelevant; below σ is a permutation over $\{1, \dots, n\}$:*

$$x \leftarrow e \mid \langle \ell_1, \dots, \ell_n \rangle \rightsquigarrow x \leftarrow e \mid \langle \ell_{\sigma(1)}, \dots, \ell_{\sigma(n)} \rangle$$

- *cancellation: this is a global rule which allows to eliminate a leakage expression that is contained in a leakage expression of another instruction:*

$$x \leftarrow e \mid \langle \ell_1, \dots, \ell_n \rangle \rightsquigarrow x \leftarrow e \mid \epsilon$$

  provided there exists another instruction of the form $x' \leftarrow e' \mid \langle \ell_1, \dots, \ell_{n+n'} \rangle$ in the program.

Note that similar rules exist for random assignments.

The correctness of leakage amplification is captured by the following statement, which is proved by induction on the derivation of $P \rightsquigarrow P'$.

Proposition 1. *If $P \rightsquigarrow P'$ and P' is non-interfering w.r.t. $\|\cdot\|$ then P is non-interfering w.r.t. $\|\cdot\|$.*

Proposition 1 has several useful consequences. For instance, security in the transition-based threshold probing model (or in the threshold probing model with glitches) entails transition in the threshold probing model. We can also recover prior results from the literature.

Proposition 2 ([4]). *A program $\mathcal{P}$ that is secure at order t in the bounded moment model if its serialization is secure at order t in the threshold probing model.*

Proposition 1 can also be used to formalize the equivalence between the models of glitches from [9] and [19].

We note that leakage amplification also opens interesting possibility for hybrid verification between models. This is typically interesting in situations involving two models: a stronger, but easier to verify, model, and a weaker, but more precise model. Embedding theorems can only be used for whole programs. In contrast, leakage amplification offers the possibility to reason in the stronger model for the part of the program where the embedding preserves security, and remains in the weaker model for the part of the program where the embedding fails. Practical applications of such hybrid techniques is an interesting direction for future work.

4 Formal verification tool

We have adapted the `maskVerif` tool to support verification for the most common models supported by `MaskIR`. In addition, we have made several improvements to the tool, both in terms of coverage and in terms of efficiency. In this section, we review the main principles of `maskVerif`, and summarize the main improvements and extensions.

4.1 Algorithms

`maskVerif` combines two main algorithms: a verification algorithm determines whether a tuple of expressions jointly depends on secrets, and an exploration algorithm which (adaptively) goes through all the possible sets of intermediate variables to analyze. Verification succeeds if the verification algorithm proves absence of leakage on all inputs given by the exploration algorithm, until there remains no further set to explore.

Verification The verification algorithm determines whether an arbitrary set of variables $\mathcal{V} = (v_1, \ldots, v_n)$ jointly depends on a secret k or not. This algorithm is organized into three rules that are successively applied to $\mathcal{V}$ as follows:

`Inputs`: $\mathcal{V} = (v_1, \ldots, v_n)$, flag $b = 0$
`Step 1`: if k is involved in the computation of at least one variable in $\mathcal{V}$, then go to `Step 2`. Otherwise return `True`.
`Step 2`: while there exists a random variable r involved exactly once in the computation of a unique variable v_i of $\mathcal{V}$, then the biggest expression e in v_i which is bijective in r is replaced by r: $e \leftarrow r$. If at least such a transformation occured, go to `Step 1`. Otherwise go to `Step 3`.
`Step 3`: if $b \neq 0$, then return `False`. Otherwise, mathematically simplify the expression of variables in $\mathcal{V}$ by developping when possible. Then, set b to one and go back to `Step 1`.

Let us take a small example to illustrate the behaviour of this algorithm. Let us consider the following set of variables $\mathcal{V} = (x_1, x_0 + r_1 + x_2, r_2)$ where r_1, r_2, x_0, and x_1 are random variables, and $x_2 = s + x_0 + x_1$ are the three shares that compose a secret s. The verification operates as follows:

1. `Step 1`: s is involved in the computation of x_2 which is itself involved in the computation of v_2, thus we go to `Step 2`.
2. `Step 2`: in the second variable $v_2 = x_0 + r_1 + x_2$, there exists a random variable r_1 such that the entire expression forming v_2 is bijective in r_1. Thus, we replace v_2 by r_1 and set $\mathcal{V}$ is now equal to (x_1, r_1, r_2). We then go back to `Step 1`.
3. `Step 1`: there is no more variable in $\mathcal{V}$ which depends on the secret s, thus we return `True`.

Exploration The exploration algorithm ensures that the main algorithm will analyze all the possible sets of at most t intermediate variables in the implementation to guarantee (at least) t-probing security. While this step is still exponential in the number of intermediate variables, `maskVerif` provides an efficient way to go through these sets. Instead of exploring all the sets containing exactly t variables, i.e., $\binom{m}{t}$ sets for m variables, the idea in `maskVerif` is to recursively verify large sets. Basically, the set of intermediate variables is split into smaller samples. In each sample, large sets are determined with hopefully much more than t variables but that are jointly independent from the secret. The verification of larger sets turns out to be practically better than the verification of all the intermediate sets of size t. When all the individual sample are verified, their intersection is handled. Doing so, the total number of sets that are actually verified is generally drastically lower than $\binom{m}{t}$ and the verification is consequently much faster.

4.2 New functionalities

In addition to verification of NI (supported in [2]) and verification of SNI (supported in [3]), the `maskVerif` tool now supports verification of threshold probing security.

Additionally, `maskVerif` supports verification in the threshold probing model with glitches. Pleasingly, only small modifications were necessary for `maskVerif` to handle the verification of hardware implementations. Essentially, the programming language `MaskIR` described above clearly indicates the leakage as observations or tuples or observations. Since `maskVerif` is already designed to verify sets which may contain more than t variables, the rest of the verification process remains exactly the same. The entire process is displayed in Section 5 and comes with a concrete example.

As an additional feature, the `maskVerif` tool now provides a simple but effective mechanism for eliminating false negatives. Originally, the `maskVerif` tool either provided a formal security proof of t-non interference or a set of potentially flawed tuples. In the latter case, users were requested to inspect manually whether the tuples represented a real attack. The `maskVerif` tool now implements a brute force algorithm that computes the joint distribution of potential flawed tuples, and verifies whether the tuple is an attack of not. This step is exact, therefore all false negatives are removed. While computing concrete distributions quickly gets very complex, it remains reasonable when it only concerns potential flawed tuples, that are not numerous in practice on the many (secure) examples we have been through.

5 Instantiation of `maskVerif` in Several Scenarios

This section aims to describe how `maskVerif` can be easily and efficiently used in the many scenarios that come from the mix between leakage models and security properties. This section is split into two parts, namely the instantiation of `maskVerif` for software scenarios and then for hardware scenarios. Examples are provided in both scenarios. For the sake of clarity, we recall here that the verification of NI and SNI properties for software implementations was already handled in the original paper of Barthe et al. [2]. In this paper, we provide extensions to additionally verify hardware implementations as well as threshold probing security for software scenario. One step further, we also provide a unified framework for software and hardware scenario based on new programming languages `MaskPC` and `MaskIR`. This section aims to give a complete overview of the current features of the update version of `maskVerif`, as illustrated in Figure 4.

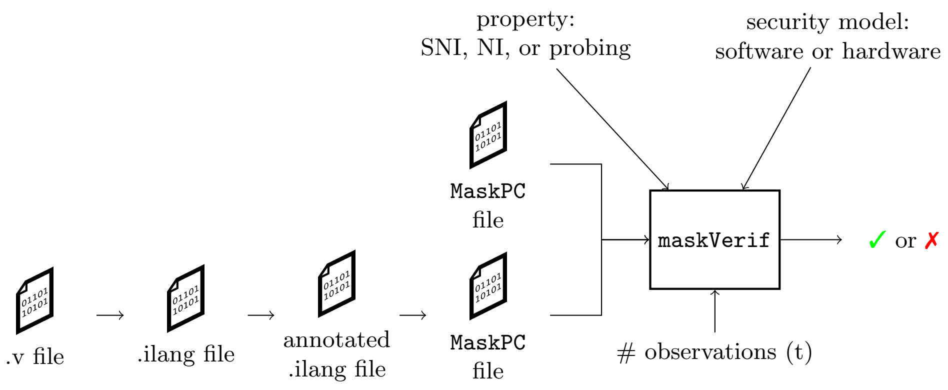

Fig. 4. Overview of software and hardware formal verifications with `maskVerif`.

5.1 Input Programming Language MaskPC

In order to unify software and hardware programs as input to `maskVerif`, we define a programming language, referred to as `MaskPC`, that describes software as well as hardware implementations. As for `MaskIR`, programs are modeled with sequence of deterministic and probabilistic assignments and are written in SSA form. Nevertheless, while no leakage information is provided, a different syntax is used to model different kind of assignment:

$$x \texttt{ := } e \quad (1) \qquad x \texttt{ = ![}e\texttt{]} \quad (2) \qquad x \texttt{ = \{}e\texttt{\}} \quad (3)$$

In the software scenario, all three assignments are equivalent and allow the adversary to learn x at the cost of one observation. In the hardware scenario, they have additional properties. Namely, (1) is for simple assignment that will generate glitches and propagate then, (2) is for storage in a register this will stop glitches propagation. The last, (3), is useful for the encoding probing model, it does not allow the adversary to learn sub-expression of e. This is exactly what is needed for the initial sharing of secret witch is assumed to be perfect in the probing model: at order 2 the sharing of a secret s will be given by the three shares $s_0 = r_0$, $s_1 = r_1$, $s_2 = s + r_0 + r_1$. The computation of the last one is assumed to be perfect, i.e. the adversary can not observe the intermediate result $s + r_0$ or she could recover s using another observation on r_0. In `MaskPC` we simply use the notation

$$s_2 = \{s + r_0 + r_1\}$$

As deeply explained below, annotated ilang files are automatically converted into `MaskPC` programs with a dedicated parser we built for this purpose. Then, the explicit characterization of the assignment allows `maskVerif` to turn `MaskPC` programs into `MaskIR` ones.

5.2 Verification of Software Implementations

Software implementations are first expected to be written in `MaskPC` programming language. Then, `maskVerif` first deterministically turns the `MaskPC` implementation into `MaskIR` implementation by interpreting the different kind affectations. Those different kinds of annotation is then used to generate the observations for `MaskIR` depending on the considered scenario. If glitches are not considered (i.e the classical scenario) then affectation (1) or (2) does not make a real difference, the leakage corresponds to e. If glitches are considered then (2) stops the propagation of glitches. For (3), the leakage is always e.

Example of MaskPC program (up) of its pending MaskIR program (middle) for classical scenario and its pending MaskIR program (down) for glitches scenario is given in Figure 5 for first-order implementations on DOM AND.

From that step, maskVerif is able to verify according to the selected property between probing security (the initial perfect sharing of secret is automatically added), non-interference, and strong non-interference, as well as the total number of authorized observations (masking order).

5.3 Verification of Hardware Implementations

In order to handle the verification of hardware implementations (i.e., with glitches) with maskVerif, we first followed the same steps than in [9]. Namely, we got use of Yosys synthesis tool [36] to generate .ilang [5] files from Verilog implementations. And on these files, we manually added some keywords to specify public variables, secret variables, output variables, and random variables. In particular, these annotations allow to specify which input wires correspond to the sharing of a secret input, as well for ouput and which input wires are random.

Our method directly starts with a Verilog masked implementation. All along this subsection, we illustrate the different steps which lead to a formal verification with the example of the DOM AND gagdet as used in [9], named here dom_and.v and graphically represented on Figure 3.

```
read_verilog dom_and.v;
hierarchy -check -top dom_and;
proc;
flatten;
opt;
memory;
opt;
techmap;
opt;
write_ilang dom_and.ilang
```

Once generated, the .ilang file is manually annotated with keywords in order to specify the public variables, the secret input variables, the secret output variables, and the random variables at the beginning of the procedure. For our example the added notations are:

```
  ## public \ClkxCI \RstxBI
  ## input \XxDI
  ## input \YxDI
  ## output \QxDO
  ## random \ZxDI
```

\XxDI is in the implementation a vector of wires (of size 2) containing the two shares of the first secret input, \XxDI correspond to second input, \QxDO contains the output shares and \ZxDI is a random input share. The ## annotations correspond to ilang comment, so they can be ignored by ilang tools. In some cases, the input of the circuit is not so naturally split into share. For example, it is possible to define the same gadget taking only one vector of secret input of size 4, say Z with the following semantic

$$Z = \{\backslash\texttt{XxDI}.[0], \backslash\texttt{YxDI}.[0], \backslash\texttt{XxDI}.[1], \backslash\texttt{YxDI}.[1]\}$$

this can be captured by using the following annotations for secret input (or output)

```
  ## input : a Z[0 2]
  ## input : b Z[1 3]
```

The annotated .ilang files can then be automatically turned into MaskPC programs. This is done by transforming the graph representation of the circuit into a linear program representation. To this end,

[5] [9] generates .json files, but we think that .ilang is more human readable and more easy to annote.

```
proc \dom_and:
 inputs : (a, [a.[1], a.[0]]), (b, [b.[1], b.[0]])
 outputs: [c.[1], c.[0]]
 randoms: r
 others : t,tp;

  tp     := b.[1] * a.[0]
  t      =![tp + r]
  tp     := b.[1] * a.[1]
  c.[1]  := t + tp
  tp     := b.[0] * a.[1]
  t      =![tp + r]
  tp     := b.[0] * a.[0]
  c.[0]  := t + tp
```

```
proc \dom_and:
 inputs : (a, [a.[1], a.[0]]), (b, [b.[1], b.[0]])
 outputs: [c.[1], c.[0]]
 randoms: r
 others : t,tp;

  tp ← b.[1] * a.[0]      |  ℓ = b.[1]*a.[0]
  t ← tp + r              |  ℓ = b.[1]*a.[0] + r
  tp ← b.[1] * a.[1]      |  ℓ = b.[1]*a.[1]
  c.[1] ← t + tp          |  ℓ = b.[1]*a.[0] + r + b.[1]*a.[1]
  tp ← b.[0] * a.[1]      |  ℓ = b.[0]*a.[1]
  t ← tp + r              |  ℓ = b.[0]*a.[1] + r
  tp ← b.[0] * a.[0]      |  ℓ = b.[0]*a.[0]
  c.[0] ← t + tp          |  ℓ = b.[0]*a.[1] + r + b.[0]*a.[0]
```

```
proc \dom_and:
 inputs : (a, [a.[1], a.[0]]), (b, [b.[1], b.[0]])
 outputs: [c.[1], c.[0]]
 randoms: r
 others : t,tp;

  tp ← b.[1] * a.[0]      |  ℓ = ⟨ b.[1], a.[0] ⟩
  t ← tp + r              |  ℓ = ⟨ b.[1], a.[0], r ⟩
  tp ← b.[1] * a.[1]      |  ℓ = ⟨ b.[1], a.[1] ⟩
  c.[1] ← t + tp          |  ℓ= ⟨ b.[1]*a.[0] + r, b.[1], a.[1] ⟩
  tp ← b.[0] * a.[1]      |  ℓ = ⟨ b.[0], a.[1] ⟩
  t ← tp + r              |  ℓ = ⟨ b.[0], a.[1], r ⟩
  tp ← b.[0] * a.[0]      |  ℓ = ⟨ b.[0], a.[0] ⟩
  c.[0] ← t + tp          |  ℓ= ⟨ b.[0]*a.[1] + r, b.[0], a.[0] ⟩
```

Fig. 5. Example of a first-order DOM AND software implementation as programmed in `MaskPC` (up) then `MaskIR` (center) in the classical scenario, and `MaskIR` (down) in the glitches scenario

we create a variable for each wire and replace each logical gate by its corresponding affectation. For example, a XOR gate with input wires x and y and output z is replaced by the instruction $z := x + y$, leading to an operation that may generate glitches. The most interesting case is for register storage or "FF"-gate which takes an input x and returns an output y which corresponds to the value of x and stop the propagation of glitches. For those gates, we generate an instruction $y =![x]$, which indicates that the propagation of x's glitches stops at that point. To linearize the graph, i.e. generate the list of instructions in the right order, we use a simple topological order on the graph. The `MaskPC` program is then given as input to `maskVerif`. The tool will automatically generate the corresponding `MaskIR` program by interpreting the affectations to define the leakage. In appendix 5.3, we provide a concrete small example to recall each step of the verification process.

From this step, our dedicated parser automatically generates a simplified file written in `MaskPC` programming language which contains the useful information for the verification. In particular, public variables, secret input variables, output variables, random variables, and local variables are first displayed. Then, each line describes a single instruction between at most two variables with two possible affectations. Symbol `:=` refers to a definition, while symbol `=![X]` refers to a definition followed by a storage in a register. As for our DOM AND example, the resulting file is displayed below. Note that the name of the intermediate variables was changed here to make the reading easier. In particular, variables `a` (originally `\XxDI`), and `b` (originally `\YxDI`) respectively split into `a.[0]` and `a.[1]`, and `b.[0]` and `b.[1]` are the secret inputs, and r (`\ZxDI`) is a uniformly distributed random variable.

```
proc \dom_and:
  publics: p1, p2
  inputs : (a, [a.[1], a.[0]]),
  (b, [b.[1], b.[0]])
  outputs: [c.[1], c.[0]]
  randoms: r

  others : tmp9, tmp11, g4, g3, g2, g1, tmp2, tmp4, tmp1, tmp3, tmp8,
           clk3, tmp10, tmp13, tmp6, clk2, tmp12, tmp15, tmp7, clk1,
           tmp5, clk0, t3, tmp14, tmp16, t0, t;

   clk3 := p2
   clk2 := p2
   clk1 := p2
   clk0 := p2
   g4 := p1
   g3 := p1
   g2 := p1
   g1 := p1
   t := !p1
   tmp1 := b.[0] * a.[0]
   tmp2 := b.[1] * a.[0]
   tmp3 := b.[0] * a.[1]
   tmp4 := b.[1] * a.[1]
   tmp5 := tmp1
   tmp6 := tmp2 + r
   tmp7 := tmp3 + r
   tmp8 := tmp4
   tmp9 := tmp6
   tmp10 = ![tmp6]
   tmp11 := tmp7
   tmp12 = ![tmp7]
   tmp13 := tmp10
   tmp14 := tmp10
   c.[1] := tmp10 + tmp4
   tmp15 := tmp12
   tmp16 := tmp12
   c.[0] := tmp1 + tmp12
```

Verifying with Glitches. Once Verilog implementations are transformed into `MaskPC` programming language, `maskVerif` first generates the corresponding `MaskIR` programs. Basically, for each instruction with symbol, the corresponding leakage ℓ is the tuple of all intermediate variables that are involved in the current computation from their last storage in a register which is symbolized with affectations with `=![]`. From such a program, `maskVerif` is able to start the analysis. As depicted before, the only change that is made to consider such hardware implementations stands in the definition of the sets to analyze. With these new observations, a significant number of intermediate variables, when observed alone, brings strictly less information to the attacker than the observation of a storage in a register. For instance, variable `a.[0]` in the above example is not expected to be observed directly since the observation of line `tmp10 = ![tmp6]` provides `b.[1]`, `a.[0]`, and `r` at the cost of a single observation. As a consequence, all the possible observations which are strictly included in other observations are removed from the set of observations for the verification phase. Obviously, all the observations which do not involve secret variables are left apart as well. Of course, it implies an increase in the size of the sets to verify, but the overhead is not high enough to compensate the smaller number of sets.

In the DOM AND example of this section, the possible observations that are recorded for the formal analysis with `maskVerif` are thus restricted only to the four following ones (with glitches):

```
tmp10 = ![tmp6]          ** ℓ =(b.[1] , a.[0] , r)
tmp12 = ![tmp7]          ** ℓ =(b.[0] , a.[1] , r)
c.[1] := tmp10 + tmp4    ** ℓ =(b.[1] * a.[0] + r, b.[1] , a.[1])
c.[0] := tmp1 + tmp12    ** ℓ =(b.[1] * a.[0] + r, b.[0] , a.[0])
```

`maskVerif` is to be called with three parameters in addition to the `MaskPC` input program, namely the security property to verify among threshold probing security, NI, and SNI, the scenario among software without glitch and hardware in the presence of glitches, and the masking order t to determine the number of authorized observations.

6 Experiments and Comparison

One of the main contributions of this paper is to demonstrate how `maskVerif` is able to efficiently handle verification of masked hardware implementations.

We thus provide in Table 1 a set of benchmarks mostly obtained from existing Verilog implementations[6]. Basically, our examples are mainly extracted from the available database provided by the authors of [9]. It gathers four different Verilog implementations of a masked multiplication. Three of them are implemented at the first masking order only, while the last one, referred to as DOM AND and designed in [21], is available up to order $t = 4$. Larger implementations are also provided, namely three S-boxes. AES S-box as designed in [21] and both versions of FIDES S-box as designed in [7] are implemented at the first order. Keccak S-box as designed in [22] is implemented from the first to the third order. To this existing set of examples, we added a few additional ones. First, Keccak S-box is also analyzed at an extra order, namely $t = 4$. Then, two versions of a different multiplication provided in [4] that we recall PARA AND in the table are verified from the first to the fourth order.

For each example, verifications are operated for the three main security properties, namely SNI, NI, and threshold probing security. For each property, verifications are performed in an hardware scenario (HW), i.e. in presence of glitches, and in a software scenario (SW), i.e. without glitch. A cross is displayed when a concrete attack is exhibited. Otherwise, the verification ends up with a formal proof. For the examples already verified in [9], we recall the features obtained by Bloem et al. in the last column, namely on threshold probing security with and without glitches, to compare their timing with ours. We use a 2.8 GHz Intel Core i7 with 16 Go of RAM running on macOS High Sierra, they use a Intel Xeon E5-2699v4 CPU with a clock frequency of 3.6 GHz and 512 GB of RAM running in a 64-bit Linux OS environment. A dash is displayed when timings are not available. The first column of the table (# obs) indicates the number of possible observations in the targeted implementation. In the software scenario, this number corresponds to the number of intermediate variables. In the hardware scenario with glitches, the number of observations is given after simplification. It is advantageously much lower than in the software scenario since observations that are included in larger ones are voluntarily erased. For instance, observing the output of a combinatorial logic set generally provides the knowledge of all its inputs. Observing each input individually is in this case non optimal and perfectly covered by the verification of the larger available set. Note that while this first column displays the number of observations n that will be further treated, verification at order t requires the analysis of $\binom{n}{t}$ tuples. For instance, the verification of Keccak S-box in the software scenario at the fourth-order requires the analysis of $\binom{450}{4} \approx 2^{31}$ tuples.

Most of the implementations presented here do not satisfy the SNI notion with glitches. In fact for those that are SNI in the SW scenario and NI in the HW scenario it is generally sufficient to store the result into a register before returning it (i.e. remove glitches from the output) to achieve SNI security in the HW scenario. This is the solution implemented for PARA AND SNI.

It is worth noting from Table 1 that as presumed earlier in this paper, our verifications with glitches often happen to be faster than the same verifications without glitches. As previously explained, this is due to the much lower number of sets to jointly analyze. Basically, the attacker can get access to more intermediate variables for the same cost via glitches, which makes the observation of individual intermediate variables needless. Furthermore, even if the sets to analyze are much bigger, their verification is not much longer. For instance the fourth-order implementation of Keccak S-box requires 1 minute and 51 seconds for a verification of the non-interference property with glitches and more than 7 minutes when the same property is verified without glitches. While this difference of timings appear in almost of the

[6] All the programs and logs are available at `https://sites.google.com/view/maskverif/home`

Table 1. Overview of hardware verification of masked circuits, we put time 0.01 when it is less of equal to 0.01

	# obs		SNI		NI		probing		probing [9]	
	HW	SW	HW	SW	HW	SW	HW	SW	HW	SW
first-order verification										
Trichina AND [35]	2	13	0.01s ✗	0.01s ✗	0.01s ✗	0.01s ✗	0.01s ✗	0.01s ✗	≤ 2s ✗	≤ 1s ✗
ISW AND [23]	1	13	0.01s ✗	0.01s	0.01s ✗	0.01s	0.01s ✗	0.01s	≤2s ✗	≤1s
TI AND [29]	3	21	0.01s ✗	0.01s ✗	0.01s	0.01s	0.01s	0.01s	≤3s	≤1s
DOM AND [21]	4	13	0.01s ✗	0.01s	0.01s	0.01s	0.01s	0.01s	≤2s	≤1s
PARA AND [4]	6	16	0.01s	0.01s	0.01s	0.01s	0.01s	0.01s	-	-
DOM Keccak S-box [22]	20	76	0.01s ✗	0.01s	0.01s	0.01s	0.01s	0.01s	≤20s	≤1s
DOM AES S-box [21]	96	571	-	-	0.08s ✗	0.3s ✗	2.3s	0.4s	≤5-10h*	≤30s*
TI Fides-160 S-box [7]	192	6657	0.2s ✗	0.2s ✗	0.3s	40s	0.3s	1s	≤1-3s*	≤1-2s*
TI Fides-192 APN [7]	128	69281	2.6s ✗	2.9s ✗	2.6s		2.5s	1m26s	≤ 5s-2h	≤2s-20m
second-order verification										
DOM AND [21]	12	30	0.01s ✗	0.01s	0.01s	0.01s	0.01s	0.01s	≤1s	≤1s
PARA AND [4]	15	30	0.01s	0.01s	0.01s	0.01s	0.01s	0.01s	-	-
DOM Keccak S-box [22]	60	165	16s ✗	0.1	0.04s	0.04s	0.02s	0.02s	≤40s*	≤10s*
third-order verification										
DOM AND [21]	20	54	0.01s ✗	0.02s	0.02s	0.02s	0.02s	0.02s	≤20s	≤4s
PARA AND NI [4]	20	48	0.01s ✗	0.01s ✗	0.02s	0.03s	0.02s	0.02s	-	-
PARA AND SNI [4]	28	53	0.02s	0.05s	0.02s	0.04s	0.02s	0.02s	-	-
DOM Keccak S-box [22]	100	290	-	46s	1.6s	2.7s	0.28s	0.25	≤25m*	≤4m*
fourth-order verification										
DOM AND [21]	30	87	0.03s ✗	0.34s	0.1s	0.15s	0.1s	0.1s	≤7m	≤2m
PARA AND NI [4]	35	75	0.01s ✗	0.01s ✗	0.02s	0.03s	0.08s	0.1s	-	-
PARA AND SNI [4]	40	85	0.3s	0.7s	0.1s	0.3s	0.1s	0.1s	-	-
DOM Keccak S-box [22]	150	450	-	6h26m	1m51s	7m36	11s	14s	-	-
fifth-order verification										
DOM Keccak S-box [22]	210	618	-	-	3h31m	-	9m44s	18m39s	-	-

verifications performed in the table, it can observed that the first-order verification of the AES S-box in the threshold probing model is longer in presence of glitches. This specific case is due to fact that the implementation is not NI, and dedicated verification of potential flawed tuples needs to be performed to ensure the threshold probing security.

Furthermore, Table 1 shows that `maskVerif` achieves better performances than the algorithm provided in [9] which requires a strong labelling with Fourier coefficients computation. For instance, the verification of the hardware first-order masked implementation of AES S-box is at the very least 7826 times much faster with our new version of `maskVerif`. In particular, note that some of the benchmarks provided for the tool of Bloem et al. only concern the verification of one secret (the ranking correspond to the fastest and the lowest verification of the secrets). They are highlighted with a symbol $*$. As a consequence, without parallelization (which we do not use in this work), these timings should probably be significantly higher. Eventually, the efficient algorithms coming with `maskVerif` and its improvements make possible the verification of higher-order masked implementations in hardware or software for concrete schemes of higher algebraic degree.

7 Conclusion

We have presented a general framework for analyzing the security of masked implementations, and adopted the `maskVerif` tool to support verification of different models. We believe that our framework and tool can be applied without any difficulty to the transition-based threshold probing model and secure multi-party computation based on additive secret sharing. We also contend that it should be direct to extend our work beyond purely qualitative security definitions, and to consider quantitative definitions that upper bound how much leakage reveals about secrets—using total variation (a.k.a. statistical) distance [18]. We believe that our work may be extended to quantitative notions, at the cost of a significant computational overhead.

Our results show that we do not have a AES Sbox implementation SNI-secure in the glitch model. If one wants to have a full AES hardware implementation secure, we need: (i) a SNI-secure implementation of the AES Sbox (which is surely possible by adding some refresh gadgets to some implementation, but the ones we have are not even NI-secure) and (ii) formally prove a composition theorem in the glitch model.

References

1. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 64–81, 2014.
2. G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, Apr. 2015.
3. G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini. Strong non-interference and type-directed higher-order masking. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 16*, pages 116–129. ACM Press, Oct. 2016.
4. G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 535–566. Springer, Heidelberg, May 2017.
5. 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, Heidelberg, Aug. 2013.
6. S. Belaïd, F. Benhamouda, A. Passelègue, E. Prouff, A. Thillard, and D. Vergnaud. Randomness complexity of private circuits for multiplication. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 616–648. Springer, Heidelberg, May 2016.
7. B. Bilgin, A. Bogdanov, M. Knežević, F. Mendel, and Q. Wang. Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In G. Bertoni and J.-S. Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 142–158. Springer, Heidelberg, Aug. 2013.
8. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-order threshold implementations. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 326–343. Springer, Heidelberg, Dec. 2014.
9. R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 321–353, 2018.
10. 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, Heidelberg, Aug. 1999.
11. J. Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In *Applied Cryptography and Network Security*, 2018. Preliminary version available as IACR eprint 2017/879.
12. J. Coron, C. Giraud, E. Prouff, S. Renner, M. Rivain, and P. K. Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, pages 69–81, 2012.
13. 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, Heidelberg, Mar. 2014.
14. J. Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In W. Fischer and N. Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 137–153. Springer, Heidelberg, Sept. 2017.
15. 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, Heidelberg, May 2014.
16. S. Dziembowski, S. Faust, and M. Skórski. Optimal amplification of noisy leakages. In E. Kushilevitz and T. Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 291–318. Springer, Heidelberg, Jan. 2016.

17. H. Eldib, C. Wang, and P. Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.
18. H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(10):1558–1568, 2015.
19. S. Faust, V. Grosso, S. M. D. Pozo, C. Paglialonga, and F.-X. Standaert. Composable masking schemes in the presence of physical defaults and the robust probing model. Cryptology ePrint Archive, Report 2017/711, 2017. `http://eprint.iacr.org/2017/711`.
20. H. Groß and S. Mangard. Reconciling d+1 masking in hardware and software. In W. Fischer and N. Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 115–136. Springer, Heidelberg, Sept. 2017.
21. H. Groß, S. Mangard, and T. Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In H. Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 95–112. Springer, Heidelberg, Feb. 2017.
22. H. Gross, D. Schaffenrath, and S. Mangard. Higher-order side-channel protected implementations of keccak. Cryptology ePrint Archive, Report 2017/395, 2017. `http://eprint.iacr.org/2017/395`.
23. 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, Heidelberg, Aug. 2003.
24. 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, Heidelberg, Aug. 1999.
25. S. Mangard, T. Popp, and B. M. Gammel. Side-channel leakage of masked CMOS gates. In A. Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, Heidelberg, Feb. 2005.
26. S. Mangard, N. Pramstaller, and E. Oswald. Successfully attacking masked AES hardware implementations. In J. R. Rao and B. Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, Heidelberg, Aug. / Sept. 2005.
27. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, Heidelberg, May 2011.
28. 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, Heidelberg, Sept. 2012.
29. S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In P. Ning, S. Qing, and N. Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, Dec. 2006.
30. 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, Heidelberg, May 2013.
31. O. Reparaz. A note on the security of higher-order threshold implementations. Cryptology ePrint Archive, Report 2015/001, 2015. `http://eprint.iacr.org/2015/001`.
32. O. Reparaz. Detecting flawed masking schemes with leakage detection tests. In T. Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 204–222. Springer, Heidelberg, Mar. 2016.
33. O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating masking schemes. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 764–783. Springer, Heidelberg, Aug. 2015.
34. 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, Heidelberg, Aug. 2010.
35. E. Trichina. Combinational logic design for AES subbyte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003. `http://eprint.iacr.org/2003/236`.
36. C. Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.
37. J. Zhang, P. Gao, F. Song, and C. Wang. Scinfer: Refinement-based verification of software countermeasures against side-channel attacks. In *Computer-Aided Verification*, 2018.