LeakageVerif: Scalable and Efficient Leakage Verification in Symbolic Expressions

Quentin L. Meunier, Etienne Pons, Karine Heydemann,

Abstract—Side-channel attacks are a powerful class of attacks targeting cryptographic devices. Masking is a popular protection technique to thwart such attacks as it can be theoretically proven secure. However, correctly implementing masking schemes is a non-trivial task and error-prone. If several techniques have been proposed to formally verify masked implementations, they all come with limitations regarding expressiveness, scalability or accuracy. In this work, we propose a symbolic approach, based on a variant of the classical substitution method, for formally verifying arithmetic and boolean masked programs. This approach is more accurate and scalable than existing approaches thanks to a careful design and implementation of key heuristics, algorithms and data structures involved in the verification process. We present all the details of this approach and the open-source tool called `LeakageVerif` which implements it as a python library, and which offers constructions for symbolic expressions and functions for their verification. We compare `LeakageVerif` to three existing state-of-the-art tools on a set of 46 masked programs, and we show that it has very good scalability and accuracy results while providing all the necessary constructs for describing algorithmic to assembly masking schemes. Finally, we also provide the set of 46 benchmarks, named `MaskedVerifBenchs` and written for comparing the different verification tools, in the hope that they will be useful to the community for future comparisons.

1 Introduction

Side-Channel Attacks (SCA) exploit the relationship between physical quantities such as time, power consumption or electromagnetic emissions and secret data manipulated by cryptographic implementations to retrieve these secret data. Since the pioneer Differential Power Analysis attack (DPA) [21], several other SCA based on power consumption or electromagnetic emissions have been shown to be very effective to recover the secret key [22]. As with the advent of the Internet-of-thing, more and more devices use cryptographic implementations and are accessible, these attacks have become a serious threat and protecting such devices against SCA has become a major concern.

Masking is a popular protection technique to thwart SCA as it can be theoretically proven secure. Its principle is to break the dependency between measurable physical quantities and manipulated secret data by making intermediate computations statistically independent from the secrets. This is achieved by splitting each secret data into $n+1$ shares, n being the masking order, such that only the recombination of all the shares allows to deduce information on the secret. A simple way to split a secret on $n+1$ shares is to use n random and uniform variables, called masks, and to obtain the last share by combining the n first shares with the secret. This combination depends on the masking type: boolean masking uses exclusive-or (xor, $\oplus$)), arithmetic masking uses addition. An algorithm for which each intermediate result is statistically independent from its secrets is said to be leakage-free at order 1. However, in order to achieve this leakage-free property, each procedure of the non-masked algorithm must be adapted to take the sharing into account. This is not a trivial task as non-linear operations w.r.t the masking scheme require special care to avoid unmasking the secrets and because leaking expressions can be hard to find by hand. Moreover, from a (proven-secure) masked algorithm to the final masked implementation, several flaws can be introduced: a correctly masked algorithm can be incorrectly implemented in software or hardware, or a correctly masked source code can lead to a leaking assembly code due to the compilation and its optimisations that can reorder computations. As a consequence, it has emerged a critical need for automatic verification methods to check the leakage-free property of software or hardware implementations in the different abstraction levels (algorithmic, C or HDL program, assembly program).

Different methods for automatically verifying the security of a masked hardware or software implementation has been proposed. There are all based on the analysis of the expressions manipulated by the program or circuit at a given abstraction level. Such methods aim to determine if the probability distribution of an expression depends on the secrets of the algorithm. If the first proposed methods relied on enumeration over all possible values, explicitly or implicitly using SMT formulations [12], [13], the more recent symbolic techniques avoid enumeration as it is not a scalable approach. Symbolic verification methods can be split into two categories: methods by *inference* [9], [10], [14], [15], [30] and methods by *substitution* [1], [2]. Approaches using an *inference* method make use of a set of rules to infer the distribution type an expression from the current operation and the distribution type of its children, going from the leaves to the root of the expression. A distribution type can usually be one of: secret dependent, secret independent, uniform, or unknown when the verification algorithm cannot conclude. Approaches using *substitution* aims to transform the expression by 1) finding a sub-expression masked by a random variable r which is not present in

- *Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France.*
E-mails: {quentin.meunier,etienne.pons,karine.heydemann}@lip6.fr

the sub-expression; 2) replacing the sub-expression by the random variable r. By iterating this find-replace strategy, if the expression is correctly masked, there should eventually no longer be any occurrence of any secret variable at the end of the substitutions: the expression is then considered leakage free. With both types of symbolic methods, the verification can fail to conclude for some expressions which are then considered as "possibly leaking". Resorting to an enumerative technique to determine the distribution type may sometimes help to conclude but this work around is limited to small expressions or variable sizes, due to the inherent non-scalability of enumeration. As a consequence, verification methods must be as precise as possible to be able to conclude for as many leakage-free expressions as possible.

Last but not least, the tools implementing such approaches must be scalable and efficient as real masked implementations require to analyze a large number of expressions that may contain several secret data and random variables.

While different leakage verification tools exist, in particular `QMVerif` [1], `SELA` [23] and `maskVerif` [15], as we show in this paper, no existing tool fulfills at the same time versatility, scalability and accuracy. The work presented in this article aims at filling this gap, by making the following contributions:

- We propose an efficient and scalable approach for verifying the first-order leakage-free property of a complex expression. We highlight the key aspects which make it possible to reach such a scalability compared to other existing tools;
- We provide an open-source implementation of this approach in a tool called `LeakageVerif`;
- We provide a thorough comparison of `LeakageVerif` with three state-of-the-art tools for verifying expressions, on a set of 46 representative masked programs, and we show that `LeakageVerif` performs better than the other tools in terms of expressiveness, scalability and accuracy;
- Finally, we also provide the first open-source collection of masked benchmarks for verification, called `MaskedVerifBench`, in the hope that they will be useful for the community. For each benchmark, we provide a description and a version for the tools `LeakageVerif`, `maskVerif` and `QMVerif` whenever possible.

The rest of the paper is organized as follows: section 2 presents some background and related works regarding masking and existing verification tools; section 3 presents our approach for verifying symbolic expression; section 4 presents the benchmarks we used and a detailed comparison between existing tools; finally, section 5 concludes and discusses future work.

2 Background and Related Works

2.1 Security Notions

Different security properties have been proposed regarding masking schemes [1], [12], [19]. Among those, the most employed property is *threshold probing security*. A program or device is threshold probing secure at a given order d if an attacker able to probe d internal values during the execution cannot learn any information on the secret. In the rest of the article, we focus on this security property, even if some stronger properties exist, such as Non-Interference (NI) or Strong Non-Interference (SNI) [1].

This article precisely focuses on first order probing security, as it is a criteria that all masked programs must verify, and as it is a basic block for higher-order verification methods [2]. Having an efficient first order verification is therefore critical.

Verification must be made according to a leakage model, which is an abstraction of real leakages. Two commonly used leakage models are: 1) the value-based leakage model, in which the leakages are the computation results, and 2) the transition-based leakage model, for which the leaked values are either any combination, or the exclusive-or, between two values consecutively hold in a shared resource (e.g. a variable, a assembly register, a bus or any hardware element). Analysing a program with expressions based on transitions requires to have an underlying implementation giving variables storing expressions, the written assembly registers, or other hardware elements, depending on the abstraction level. Even if low-level descriptions enable more accurate results w.r.t. the real leakages, this knowledge is not always available. Most importantly, once an implementation is known, it is easy to build expressions modeling transitions to match this implementation and transition-based leakage model. Moreover, a tool able to verify leakages in the value model can easily make the verification in the transition model. For this reason, we only consider leakages in the value model in this article.

2.2 Masking Schemes

If transforming a program in a masked equivalent is usually not an automatic process, there exists some proven constructions for small boolean functions, most especially in hardware, which allow to design masked circuit fragments (called gadgets). The advantage of such constructions is that they are guaranteed to be well masked, but their composition is not, and often requires registers between gadgets.

Notable masking schemes include the ISW one, presented in [19], which consists in decomposing a circuit in AND, NOT and XOR gates, while providing an d-order threshold probing secured implementation for each of these gates. However, this scheme does not consider glitches, which are transient values in hardware due to different propagation delays between wires and gates, and which can affect a hardware implementation. The Threshold-Implementation (TI) scheme is proposed in [24] for boolean functions. The claim made by the authors is that generating random or pseudo-random value is costly, and therefore should be avoided. The scheme does not use such internal random values, but instead requires more shares and gates for achieving the same level of security. In 2015, Reparaz *et al.* [26] try to unify the previous constructions, and proposes a General Masking Scheme (GMS) model based on four layers for masking boolean functions. The gain compared to TI is a reduction in complexity for first order resistance (at the price of using more randomness), and a gain in security for higher orders. Finally, Gross *et al.* [17] present Domain-Oriented Masking (DOM), which uses a similar structure to GMS, but achieves an identical level of security with a reduced sharing order, d-order threshold probing security with d shares for the AND function. This is done by inserting registers inside the computation in order to stop glitch propagation.

As opposed to these general schemes, some masked programs have to be written by hand, what is especially true for software implementations. For such programs, there is no guaranty that every intermediate expression is leakage-free, and an automatic verification is advised. Moreover, even with a leakage-free description at the algorithmic level, an implementation of this algorithm can exhibit leakages due to the lower-level construct, e.g. the implementation of the abstract Galois field multiplication. This is all the more true at assembly level, what shows the need to be able to prove masking programs at different abstraction levels.

2.3 Masking Verification

Verifying that the distribution of a masked expression is independent from the secret values can be done with an exhaustive approach, by enumerating all possible secret values, and for each combination of secret values, by computing the distribution of the expression for every possible mask value. If the expression is well masked, all obtained distributions must be identical.

The problem of such an approach is that the time required grows exponentially with the number of bits in all variables. Symbolic methods, on the other hand, reason on variable types and operators to deduce information on the expression. For example, given a secret variable `k` and a mask `m` on n bits, we can deduce that the expression `k` $\oplus$ `m` is leakage-free for every value of n. In most cases, however, the size of each variable or constant must be provided to perform the verification.

The first substitution algorithm for verifying the absence of leakage in an expression was described in [2]. The property verified is threshold probing security for any order d. It states that the joint distribution of any set of d expressions is independent from the secrets values contained in the expressions. This algorithm was first implemented in the EasyCrypt tool [4], but not made accessible. This algorithm was then implemented for the verification of hardware circuit in a tool called `maskVerif` [1]. The latter takes as input an annotated verilog file containing the circuit to verify. The strength of `maskVerif` is to verify efficiently higher-order masked gadgets, and for three security properties: threshold probing, NI and SNI.

A substitution method was also used in the tool `SELA` [23], which is a python library with dedicated constructs for creating and verifying symbolic expressions. The goal of `SELA` is to provide an easy to use framework for writing and analysing symbolic expressions at different abstraction levels. The authors show the interest of the approach by verifying algorithmic codes as well as assembly codes or circuits. In that sense, it is somewhat similar to the goal of `LeakageVerif`. However, `SELA` suffers from important scalability issues which prevents it from being used in many real applications. This can be seen for example on its verification of the AES, which is limited to two rounds.

Another approach, presented in [9], [10] introduced symbolic analysis based on inference. Inference permits reusing the results of sub-expressions for determining the result of the current expression, while the substitution approaches must start from scratch for each new expression, even if it is a combination of already analysed expressions. The advantage of this property seems clear since algorithms and circuits can be decomposed as a succession of operations re-utilizing the results of previous operations. On the other hand, the inference rules in [9] could not conclude in some cases contrary to the substitution approach. This inference technique was used as a basis in several works: it was first implemented in a somewhat similar way in a tool called SCInfer [30], then improved with more precise rules in the QMSInfer tool [15]. Support for arithmetic operators was added in the `QMVerif` tool in [14], along with some improvements. Finally, support for function calls was added in [25] and the tool was made accessible.

2.4 Limitations of Existing Tools

The `maskVerif`, `QMVerif` and `SELA` tools are accessible along with their source code. We analysed them to the extent of our understanding and tried to use them as much as possible. It came out that each of them has limitations when used as a basis for verifying general masked expressions, such as expressions resulting from C codes or assembly codes.

maskVerif. `maskVerif` is tool written in Caml designed for the verification of circuits, but proposes a *software scenario* for the verification of algorithms, in which glitches are not considered, and in which the sequential aspect of the program is simulated using a register-like behaviour: each expression computed by the program is written into a register. However, it is still not very well adapted for some benchmarks. In particular, it lacks support for arithmetic operations like the addition or array accesses. It also lacks support for arbitrary size variables and expressions, since the only possible sizes for variables are 1, 8 and 32 bits, and there is no bit concatenation/extraction operations, what makes this tool hardly suited to the analysis of assembly code. Moreover, as it is designed for verifying circuits, it is not possible to verify the distribution of a word composed of several bits, what forced us to use tricks for some programs, namely to combine the different bits of a word in a way they don't simplify. Finally, still linked to the hardware orientation of the tool, there is no way of expressing a basic control flow besides function calls.

We also encountered two errors, reported to the authors: an incorrect result in some cases (leaking expression proven leakage-free, see appendix A), and incorrect expressions when using function calls, forcing us to manually inline functions instead.

QMVerif. As a tool for verifying algorithms, `QMVerif` does not have some of the problems of `maskVerif`. It is a tool written in C++, which is able to verify expressions on words and supports arithmetic operations. However, variables are all on 8 bits, and there is no way for having a variable or an expression on a different size. If some algorithms behave the same way independently from variable size (in case only bitwise operations are used), this is often not the case. Moreover, some implementations can also mix different sizes together, e.g. the AES which can manipulate both bytes and words in some implementations for performance reason. Besides, there is no support for basic control flow like loops. Additionally, it uses as input an ad-hoc format in which expression must be written using a single operator per line.

Besides, although the authors of [25] claim their tool to be open-source, the version of the tool we have access to does not have the characteristics presented in the article: it has no support for functions and assumptions, and no support for additional rules, called transformation oracles.

Additional limitations include the fact that the parser does not report some errors and that some operators which are supposed to be supported, such as shifts, are also never returned by the parser. Finally, there are some limitations due to hard coded values, in particular regarding the number of masks during enumeration.

SELA. `SELA` does not have many of the previously mentioned problems. The fact that it is provided as a python library allows it to benefit from python operators and its control flow constructions. Its syntax based on z3py [27] allows it to manage expressions of arbitrary size. Finally, an interesting aspect of `SELA` comes from the fact that by changing the definition of symbolic variables with constant values, it is possible to check the results, and thus the functionality of the program implemented. This is a very useful feature, which is not possible with `maskVerif` and `QMVerif`.

The major limitation of `SELA` comes from the fact that when two expressions are combined with an operator, they are entirely copied to create a new expression. This results in an exponential growth of the expression size in memory when some sub-expressions are used several times, e.g. in the mix-columns part of the AES. As a consequence, it has a limited scalability in the number of operations.

3 LeakageVerif

This section presents our new substitution approach for checking secret independence in symbolic expression, implemented in a tool called `LeakageVerif`. Its goal is to provide a user-friendly, scalable, accurate, versatile and open-source verification process, while overcoming the limitations of other existing approaches and tools. We claim that it is currently the best tool for verifying the absence of leakage in expressions in the first order threshold probing leakage model.

3.1 Overview

Inspired by `SELA`, `LeakageVerif` comes as a python library, and offers constructions for variables, constants and expressions by overloading standard python operators. Variables are symbolic and must have a type among `Secret`, `Public` or `Mask`. Variables and constants must have a defined size in bits. Threshold probing security can be verified using the function `checkTpsVal` for value based leakage, and `checkTpsTrans` for transition-based leakage. Evaluation using exhaustive enumeration is also possible, but will not be considered in the rest of the article as it is not a scalable approach.

```
# 8-bit variable named 'm0' of type Mask
m0 = symbol('m0', 'M', 8)
# 8-bit variable named 'm1' of type Mask
m1 = symbol('m1', 'M', 8)
# 8-bit variable named 'k0' of type Secret
k0 = symbol('k0', 'S', 8)
# 8-bit variable named 'k1' of type Secret
k1 = symbol('k1', 'S', 8)
# expression computation
exp = (m0 ^ k0) & (m0 ^ m1 ^ k1)
# check for leakage in the expression value
res = checkTpsVal(exp)
```

Figure 1. Example of `LeakageVerif` program verifying threshold probing security for the expression `(m0` $\oplus$ `k0) & (m0` $\oplus$ `m1` $\oplus$ `k1)`

Figure 1 shows a code fragment of a `LeakageVerif` program, verifying the threshold probing security of the expression `(m0` $\oplus$ `k0) & (m0` $\oplus$ `m1` $\oplus$ `k1)`.

`LeakageVerif` implements all the basic boolean operations (not, and, or, xor), arithmetic operations (addition, subtraction, multiplication, multiplication in finite field), and shift operations (logical shift left, arithmetic and logical shift right) on variables of any size. It also supports array indexation, and operators for performing bit manipulations: concatenation of expressions, extraction of a bitfield in an expression, signed and unsigned extensions of expressions. These operations allow to support the vast majority of masked programs, in particular as we did not encounter any other operations in all the benchmarks we found.

3.2 Design and Implementation Challenges

The original substitution algorithm for proving threshold probing security is described by Barthe *et al.* in [2]. Each mask variable can be selected at most once as a substitution basis in order for the algorithm to be correct. A variant of this algorithm, given in Algorithm 1, has been recently published in a technical report [3]. This variant notably removes the condition that a mask `m` selected for a substitution should not appear anywhere else in the entire expression. The mask `m` must only not appear anywhere else in the masked sub-expression `e + m` that will be replaced by `m` (in Step 2). This variant enables to perform a substitution with a mask which has more than a single occurrence in an expression. As shown in [3], this allows to conclude successfully in cases the previous algorithm failed.

However, this opens a new challenge. During the substitution using a mask `m`, all the occurrences of `m` (but the one in the `e + m`) must be replaced with the expression `m + e`. First, this can make the expression grow very quickly. Second, as illustrated in Figure 2, choosing the masks in an incorrect order can lead to miss the ability to conclude. The heuristic to select a mask and which of its occurrences to replace is therefore critical for avoiding a dramatic increase in the expression size and for verification accuracy. The given description does not address this aspect while it is a key point in the algorithm.

Besides, independently from the mask selection heuristic, making a good choice requires first to have the knowledge of all masks and masks occurrences. Yet, this should not mean to go through the entire expression before each substitution, as this would prevent scalability.

Algorithm 1 Substitution algorithm for verifying threshold probing security, from [3].

```
procedure ThresholdProbingSecurity($e$)
    Inputs: tuple of expressions $V = (v_1, ..., v_n)$, flag simplified = 0, set of masks $M = \emptyset$
    Step 1: if a secret $k$ is involved in the computation of at least one expression in $V$ then go to Step 2. Otherwise return True.
    Step 2: while there exists a mask $m \notin M$ involved in the computation of an expression $v_i$ of $V$, then find a sub-expression $e$ in $v_i$ such that $m \rightarrow e + m$ is bijective and substitute $m$ by $e + m$ in all expressions. Extend $M$ with $\{m\}$.
    If at least such a transformation occurred, go to Step 1. Otherwise go to Step 3.
    Step 3: if simplified $\neq 0$, then return False. Otherwise, mathematically simplify the expressions in $V$. Then, set simplified to one and go back to Step 1.
```

Finally, regarding simplification (Step 3), [23] has shown that in order to increase scalability, simplifications should be made after each substitution and not only once after the first failure. However, simplification is challenging: it should not be source of expression size increase nor limits accuracy of the verification.

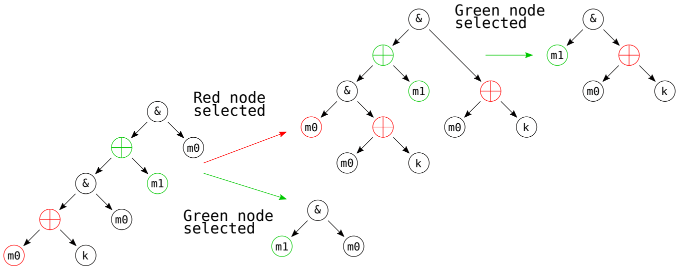
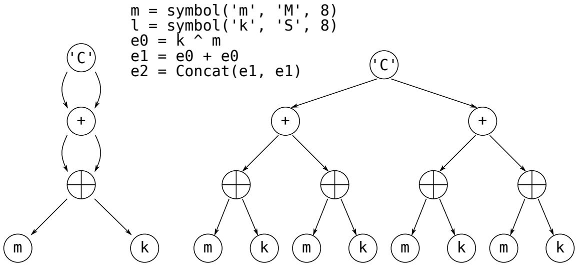


Figure 2. This example illustrates the importance of the order in which masks are selected when performing the substitutions. `m0`, `m1` are masks, `k` is a secret. If the mask `m0` with 3 occurrences is chosen for the first substitution (occurrence below the red ⊕ node), a secret occurrence will remain even after the second substitution. On the other hand, if `m1` with a single occurrence is chosen first (green ⊕ node), it allows to conclude directly after this substitution since there is no more secret occurrence.

This leads us to define some challenges that an efficient and scalable verification method should address.

- **Selection heuristic.** The mask selection heuristic must be specified, for reproducibility concern. For scalability reason, it must avoid as much as possible an increase in expression size without sacrificing accuracy.
- **Simplification.** Having a fast and effective simplification procedure is critical in order to reach both accuracy and scalability, as it helps limiting the size of the expression, and suppresses ineffective variable occurrences.
- **Information caching.** To prevent avoidable traversals of sub-expressions, which is a required condition for scalability, as much information as possible should be cached, especially regarding variables used in the sub-expressions (e.g. nature and number of occurrences).
- **Efficient algorithms.** All algorithms involved in the verification process should avoid as much as possible expression or graph traversal, and create or make use of cached information.
- **Memory representation.** The graph data structure of an expression impacts the solution to the previous points in addition to the required memory footprint. Figure 3 shows the two possible memory representations of a simple expression. As a real example, let us consider the expressions in the mix-columns part of the AES: they contain several common sub-expressions. Using the left representation will result in a graph having a number of nodes equal to the number of operations in the AES, while using the right representation will result in the number of nodes growing exponentially with the rounds.
  It seems clear that only the memory representation allowing several parents is scalable with the number of operations. Note that `SELA` uses the right representation, leading to an incorrect scalability claim by the authors.
  The graph representing an expression should thus be able to contain several edges to the same sub-expression, in order to avoid the replication of sub-graphs. We can note that using such a representation removes the possibility to modify the graph, as modifying a sub-expression when coming from one parent would modify it incorrectly for all the other parents.



```
m = symbol('m', 'M', 8)
l = symbol('k', 'S', 8)
e0 = k ^ m
e1 = e0 + e0
e2 = Concat(e1, e1)
```

Figure 3. Two different memory representations for an expression using several occurrences of a sub-expression. The representation on the left uses node alias and allows a node to have several predecessors. The representation on the right is a tree and duplicates internal nodes.

In the following, we present the core of `LeakageVerif` and how `LeakageVerif` addresses these challenges, via adapted representations and design choices.

3.3 Verification Algorithm

In this section, we first give some terminology before presenting the main verification algorithm.

3.3.1 Terminology

In the following we call *expression graph* the n-ary graph $G = (V, E, \mathcal{S}, \mathcal{M}, r)$ representing an *expression* to verify. r is the root node of the graph, V the set of nodes, $E \subset V \times V$ the set of edges, $\mathcal{S} \subset V$ is the set of secret variables and $\mathcal{M} \subset V$ the set of mask variables. Each node `n` in V is either an operator node, either a variable node or a constant node (and is a leaf in both last cases). All nodes but the root can have several parent nodes, simply called parents in the following.

Each node appears only once in an expression graph. We call *number of occurrences* of a node the number of times this node is encountered when exploring all the paths starting from the root. For a variable node this corresponds to the number of times the variable appears in the corresponding expression.

An operator node `n` is a masking operator node if the corresponding operation is a masking operation, i.e. $\oplus$ or `+`. A mask occurrence of a mask `m` is said to be *masking* if there exists a masking operator node, when going up in the graph, whose value is bijective w.r.t. the value of `m`. Such a masking operator node is called a *masking node*. For a given mask, we call *base masking node* the first masking node encountered on a path going up in the graph. Figure 4 illustrates these different terms.

The verification algorithm searches for masking nodes. When a substitution occurs, the selected masking node is called the *node to replace*. It is replaced with the node of `m`, while all the other occurrences of `m` are replaced with the sub-expression graph starting at the masking node.

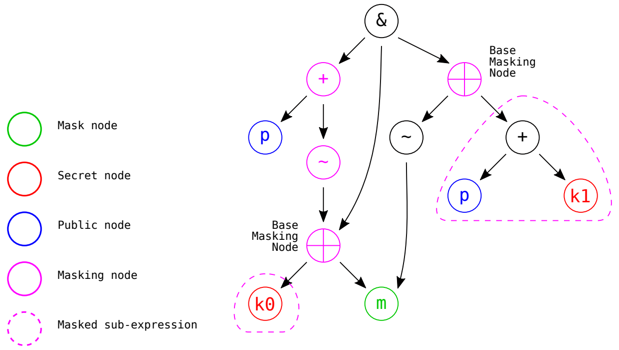


Figure 4. Expression graph example. In this example, the mask `m` has three occurrences and two parents. The left $\oplus$ node has two occurrences, while the right $\oplus$ node has one occurrence.

3.3.2 Main verification algorithm

The main algorithm used for verification, adapted from [23] is given in Algorithm 2. Substitutions are made until there is no more secret occurrence in the expression, until a mask is masking the whole expression, in which cases the value *True* is returned, or until no more replacement can be made, in which case the value *False* is returned. A simplification is made at the beginning and after each replacement. When the algorithm returns *True*, it guarantees that the expression is secret independent. This implies that secret dependent expressions are always detected by the return value *False*. The return value *False* is also returned when the algorithm fails to conclude for a secret independent expression (false positive). Therefore, the return value *False* only indicates a possible leakage. Figure 5 illustrates the principle of the main procedure.

While entirely transparent to the user, the verification can either be made at the word level, or at the bit-level. This depends on the 1) operations present in an expression (e.g. expressions containing only bitwise operators are never verified at word-level) 2) on the previous verification results of this expression's sub-expressions (e.g. when a word level verification has failed on an expression `e`, it will not be tried on expressions containing `e` as a sub-expression), and on the result given by the first call to `ThresholdProbingSecurity` (e.g. when word-level verification fails, as shown by Ines et al [9], bit-level verification can sometimes conclude). At bit-level, all variables are decomposed in as many 1-bit variables as the number of bits they contain, keeping the same nature, and all expressions are decomposed as the concatenation of 1-bit expressions. For word expressions as for decomposed expressions, the same function `ThresholdProbingSecurity` is called on the expression root.

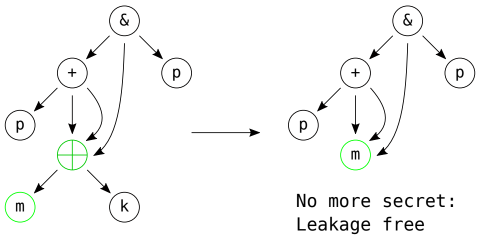


Figure 5. `m` is a mask, `k` is a secret, `p` is a public variable. The mask `m` has a single occurrence in the graph. The $\oplus$ node above `m` masks the expression `m` $\oplus$ `k`, and is replaced with `m`. The graph contains no more secret after the substitution.

3.4 Information Caching via Node Attributes

`LeakageVerif` was designed to keep the maximum information needed for the verification algorithm in each node. In particular, for some notions defined in the previous section and additional ones needed for the algorithms involved in the verification we cache them using a set node attributes. These attributes are inferred from node to node. They are computed once for each node of an expression graph using some inference rules during node creation. In this section we present these node attributes and inference rules.

Let us first define a set of simple predicates used by the inference rules:

- `maskingOp(n)` $\rightarrow \mathbb{B}$: states that `n` is a masking operation (i.e. $\oplus$ or `+`).
- `preservingOp(n)` $\rightarrow \mathbb{B}$: states that the operator node `n` preserves the masking property of its single child. It is true for unary bijective operations `~` and `A` (bijective array access).
- `parent(p, n)` $\rightarrow \mathbb{B}$: states that node `p` is a parent of the node `n`.
- `children(n)` $\rightarrow \mathcal{P}(V)$: returns the set of children of the node `n`.
- $\mathcal{V}$(`n`) $\rightarrow \mathcal{P}(V)$: returns the set of nodes of the expression sub-graph rooted at `n`.

We now define the following node attributes that we compute using the inference rules presented in Figure 6. Each attribute is given as a function but is actually attached to the expression graph rooted at node `n`, which is passed as the first parameter of the function.

- `preservedMask(n, m)` $\rightarrow \mathbb{B}$: states that the value of the node `n` is bijective with the value of the node `m`, while there is no masking operator in the graph between `m` and `n`. For example, in the expression `~m`, `m` is a preserved mask in the '~' node. Its default value is false. Informally, preserved masks are masks which have not yet masked an expression but can make the next masking operator node encountered a masking node. Rule `R1a` sets this attribute for unary bijective operator node whose child is a mask

Algorithm 2 Algorithm for verifying threshold probing security.

```
Require: nodeIn is the root of the expression to analyse
Ensure: False is returned if the distribution of the expression nodeIn is dependent from a secret it contain. Otherwise, True is
    probably returned.
procedure ThresholdProbingSecurity(nodeIn)
    n ← simplify(nodeIn)
    masksTaken ← set()
    while True do
        if secretVarOcc(n, .) = 0 then
            return True                                                    ▷ No more secret
        if maskedBy(n, ., .) then
            return True                                  ▷ A mask is masking the current node
        mask, nodeToReplace ← SelectMask(n, masksTaken)   ▷ mask is the mask node masking the node to replace
        if mask = None then
            return False
        masksTaken.add(mask)
        n ← GetReplacedGraph(n, mask, nodeToReplace)
        n ← simplify(n)
```

while rule R1b sets it for unary bijective operator node which is a parent of a node which preserves the masking property of a mask.

- maskedBy(n, m, bmn) → $\mathbb{B}$: states that the mask m is masking the expression graph whose root is n via the base masking node bmn. Its default value is false. Rules R2a, R2b, R2c and R2d set this attributes in the respectively following case: a) n is a masking operator and has m as a child and m has no occurrence in any other child of n; b) n is an unary bijective operator and its child is masked by m; c) n is a masking operator and has one child preserving the mask m while m has no occurrence in any other child of n; d) n is a masking operator and has one child masked by m via bnm while m has no occurrence in any other child of n.
- maskingMOcc(n, m, bmn, mn) → $\mathbb{N}$: returns the number of masking occurrences, in the expression graph whose root is n, of the mask m via the base masking node bmn and the masking node mn. Its value is set to 1 by rules R2a, R2b and R2c under the same (exclusive) conditions as explained for the maskedBy attribute. For other cases, it is set by the rule R3 as the sum of the masking occurrences of m via the base masking node bmn and the masking node mn in each child of n.
- otherMaskOcc(n, m, p) → $\mathbb{N}$: returns the number of non masking occurrences, in the expression graph whose root is n, of the mask node m for which its parent is p. Its default value is 0. It is set to 1 by rules R4a for any parent of a mask node that is not a masking operator node, and by rule R4b for masking operator nodes being a parent of m when occurrences of m exist in other children. It is computed for other nodes using rule R4c.
- noOtherMaskOcc(n, m, c) → $\mathbb{B}$: states that there is no occurrence of m in the expression graph rooted at n other that the ones in the sub-graph rooted at c. Its value is false for leaves and is otherwise computed by rule R5 that checks that there is no occurrence of the mask m in all the children of n but c.
- secretVarOcc(n, s) → $\mathbb{N}$: returns the number of occurrences of the secret variable s in the expression graph whose root node is n. Its default value is 0. It is changed by the rule R6a if n is a secret node or by the rule R6b for nodes that are not a leaf.

All these attributes are implemented in LeakageVerif mostly using python dictionaries. If the data structures required for implementing these attributes do have a memory impact, this impact remains globally reasonable on the total memory footprint. More importantly, these data structures remain scalable with the number of nodes in the expression. Moreover, they allow to have directly access to all the masking and non-masking occurrences of any mask from the root node of the expression. These precomputed attributes are thus an important implementation key point for scalability.

3.5 Selection Heuristic and Substitution Algorithm

3.5.1 Selecting a Mask Occurrence

The algorithm of the SelectMask procedure that chooses the substitution to make is given in Algorithm 3. The goal of this procedure is to choose a node in the expression graph (selectedMaskingNode) which is masked with a mask variable (selectedMask), in order to replace this node with the mask variable itself. As mentioned in section 3.2, each mask variable can be selected only once. Therefore, when all masks have already been selected, the selection process fails. It also fails when there is no masking mask occurrence in the expression. This is a 3-step process:

1) Select the mask variable to use for the substitution. We choose the mask variable m with at least a masking occurrence which has the minimum number of parents in the graph. Note that it is not necessarily the minimum number of occurrences in the expression. This choice is made because for every parent of m other than the one corresponding to the node to replace, m must be replaced with the whole expression. Thus, choosing a mask variable with a high number parents will result in the expression growing quickly, and having potentially more and more variable occurrences. This strategy is illustrated in the introductory example in Figure 2.
2) Once the mask variable is selected, one has to choose the base masking node to replace among all such nodes, which are the masking parents of this mask. We select the base masking node with the highest number of occurrences in the expression, as this choice will have the highest impact on the size of the resulting expression

$$\frac{\mathtt{preservingOp(n)} \land \mathtt{parent(n,m)}}{\mathtt{preservedMask(n,m)}}\;(\mathtt{R1a}) \qquad \frac{\mathtt{preservingOp(n)} \quad \mathtt{c} \in \mathcal{V}(\mathtt{n}) \quad \mathtt{parent(n,c)} \quad \mathtt{preservedMask(c,m)}}{\mathtt{preservedMask(n,m)}}\;(\mathtt{R1b})$$

$$\frac{\mathtt{maskingOp(n)} \quad \mathtt{parent(n,m)} \quad \mathtt{noOtherOcc(n,m,m)}}{\mathtt{maskedBy(n,m,n)},\ \mathtt{maskingMOcc(n,m,n,n)} := 1}\;(\mathtt{R2a}) \qquad \frac{\mathtt{preservingOp(n)} \quad \mathtt{c} \in \mathcal{V}(\mathtt{n}) \quad \mathtt{parent(n,c)} \quad \mathtt{maskedBy(c,m,bmn)}}{\mathtt{maskedBy(n,m,bmn)},\ \mathtt{maskingMOcc(n,m,bmn,n)} := 1}\;(\mathtt{R2b})$$

$$\frac{\mathtt{maskingOp(n)} \quad \mathtt{c} \in \mathcal{V}(\mathtt{n}) \quad \mathtt{parent(n,c)} \quad \mathtt{preservedMask(c,m)} \quad \mathtt{noOtherOcc(n,m,c)}}{\mathtt{maskedBy(n,m,n)},\ \mathtt{maskingMOcc(n,m,n,n)} := 1}\;(\mathtt{R2c})$$

$$\frac{\mathtt{maskingOp(n)} \quad \mathtt{c} \in \mathcal{V}(\mathtt{n}) \quad \mathtt{parent(n,c)} \quad \mathtt{maskedBy(c,m,bmn)} \quad \mathtt{noOtherOcc(n,m,c)}}{\mathtt{maskedBy(n,m,bmn)},\ \mathtt{maskingMOcc(n,m,bmn,n)} := 1}\;(\mathtt{R2d})$$

$$\frac{\mathtt{c} \in \mathcal{V}(\mathtt{n}) \quad \mathtt{parent(n,c)} \quad \mathtt{maskingMOcc(c,m,bmn,mn)}}{\mathtt{maskingMOcc(n,m,bmn,mn)} := \sum_{\mathtt{u} \in \mathtt{children(n)}} \mathtt{maskingMOcc(u,m,bmn,mn)}}\;(\mathtt{R3})$$

$$\frac{\mathtt{parent(n,m)} \quad \neg\mathtt{maskingOp(n)}}{\mathtt{otherMaskOcc(n,m,n)} := 1}\;(\mathtt{R4a}) \qquad \frac{\mathtt{parent(n,m)} \quad \mathtt{maskingOp(n)} \quad \neg\mathtt{noOtherOcc(n,m,m)}}{\mathtt{otherMaskOcc(n,m,n)} := 1}\;(\mathtt{R4b})$$

$$\frac{\neg\mathtt{parent(n,m)}}{\mathtt{otherMaskOcc(n,m,p)} := \sum_{\mathtt{c} \in \mathtt{children(n)}} \mathtt{otherMaskOcc(c,m,p)}}\;(\mathtt{R4c})$$

$$\frac{\mathtt{parent(n,c)} \quad \forall \mathtt{cc} \in \mathtt{children(n)}, \mathtt{cc} \neq \mathtt{c},\ \forall \mathtt{p} \in \mathcal{V}(\mathtt{cc}),\ \mathtt{otherMaskOcc(cc,m,p)} = 0 \land \forall \mathtt{mn} \in \mathcal{V}(\mathtt{cc}), \forall \mathtt{bmn} \in \mathcal{V}(\mathtt{cc}),\ \mathtt{maskingMOcc(cc,m,bmn,mn)} = 0}{\mathtt{noOtherOcc(n,m,c)}}\;(\mathtt{R5})$$

$$\frac{\mathtt{s} \in \mathcal{S}}{\mathtt{secretVarOcc(s,s)} := 1}\;(\mathtt{R6a}) \qquad \frac{}{\mathtt{secretVarOcc(n,s)} := \sum_{\mathtt{c} \in \mathtt{children(n)}} \mathtt{secretVarOcc(c,s)}}\;(\mathtt{R6b})$$

Figure 6. Inference rules for node's attributes. By convention, *n*, *c* and *p* are operator nodes, *m* is a mask node, *s* is a secret node, while *cc* can be of any node nature.

graph after the replacement. Figure 7 illustrate this step on a simple example.

3) Finally, once a base masking node has been selected, if its parent and ancestors are masking or preserving operators, it can be beneficial to replace a node higher up in the expression in order to replace the biggest possible sub-graph. However, this can only safely be made, *i.e.* without making the expression more complex, if the number of occurrences of the new node to replace is equal to the one of the base masking node. Figure 8 illustrates this step.

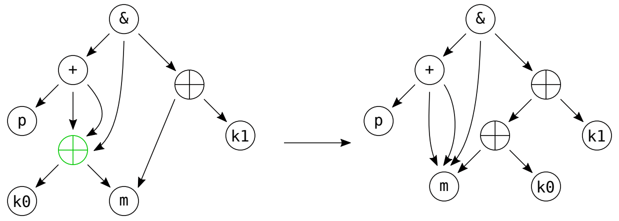


Figure 7. Illustration of step 2: Choosing the base masking node to replace. `m` is the only mask variable, `k`, `k0` are secrets, while `p` is a public variable. There are two occurrences of possible nodes to replace in the graph. We select the green one, with three occurrences in the expression.

3.5.2 Substitution by Creating a New Graph

The substitution consists in replacing one or several nodes in an expression graph. Since existing nodes cannot be modified, all the nodes above a replaced node must be rebuilt, up to the root. This is not the case for inference methods, for which the expression graph is never modified and never has to be traversed.

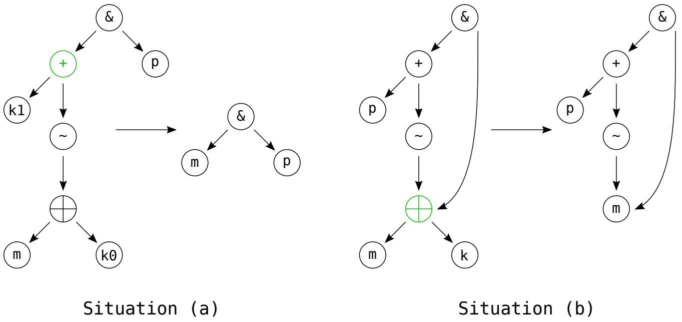


Figure 8. Illustration of step 3: Selecting the node to replace. `m` is a mask, `k0`, `k1` are secrets, while `p` is a public variable. In both situations, there is only one possible base masking node at the end of step 2i (the ⊕ node). In situation (a), the node to replace is higher than the base masking node to cover the occurrence of `k1`. It is possible since ∼ and ⊕ are bijective operators. In situation (b), the base node to replace at the end of step 2, the green ⊕ node, is the final node to replace because the ∼ node above it has less occurrences than this node. The strategy allows to conclude in both cases.

Algorithm 4 shows the graph traversal performing the required replacements after a substitution has been decided. In this algorithm, the variable *newNodes* is a map which associates a newly created node to the corresponding node in the old graph. There are several cases when going through the current node's children: if a corresponding node has already been created in the new graph, we use this new node as a new children (l. 7); if the child is the node to replace, it is replaced with the selected mask node (l. 9); if it is the selected mask node, it is replaced with the node to replace (l. 11). Finally, if the selected mask does not appear in the child, the original child is used as the new child (l.

Algorithm 3 Algorithm for choosing a mask occurrence for a substitution

```
Require: n is the root node; masksTaken is the set of masks for which a substitution has already occurred
Ensure: Chooses the mask m with at least a masking node which minimizes the number of nodes parents of m; for this m,
    chooses the base masking node to replace (selectedBMN) with the highest number of occurrences in the expression; for this
    selectedBMN, chooses the node to replace with the maximum distance for the same number of occurrences
procedure SELECTMASK(n, masksTaken)
    minNbParents ← MAXVAL
    selectedMask ← None
                                ▷ Step 1: selecting the mask which minimizes the number of nodes parents of m
    for m such that maskingMOcc(n, m, ., .) > 0 do
        if m ∈ masksTaken then
            continue
        nbParents ← Card({bmn | maskingMOcc(n, m, bmn, .) > 0} + Card({p | otherMaskOcc(n, m, p) > 0})
        if nbParents < minNbParents then
            minNbParents ← nbParents
            selectedMask ← m
    if selectedMask is None then
        return None, None
            ▷ Step 2: choosing the base masking node to replace with the highest number of occurrences in the expression
    maxCount ← 0
    selectedBMN ← None                                          ▷ Selected base masking node to replace
    for bmn such that maskingMOcc(n, selectedMask, bmn, .) > 0 do     ▷ bmn is a base masking node
        if maskingMOcc(n, selectedMask, bmn, bmn) > maxCount then
            maxCount ← maskingMOcc(n, selectedMask, bmn, bmn)
            selectedBMN ← bmn
          ▷ Step 3: Choosing the masking node to replace with the maximum distance for the same number of occurrences
    maxDist ← -1
    nodeToReplace ← selectedBMN
    for mn such that maskingMOcc(n, selectedMask, selectedBMN, mn) > 0 do
                                        ▷ mn is a masking node with base masking node selectedBMN
        dist ← distance(selectedBMN, mn)          ▷ distance(a, b) is the number of nodes between a and b
        if maskingMOcc(n, selectedMask, selectedBMN, mn) = maxCount and dist > maxDist then
            maxDist ← dist
            nodeToReplace ← mn
    return selectedMask, nodeToReplace
```

16); in the other case, a recursive call is made to construct the corresponding graph (l. 13). Eventually, the new node is created with the same operator as the previous one, and with the new list of children (l. 17).

Note that except when a substitution occurs, inducing the creation of a new graph, the graph expression is never traversed.

3.6 Simplification and Equivalence Comparison

A simplification is made at the beginning and after each replacement in the main verification algorithm. As previously explained, it is challenging and important for both scalability and accuracy. We present in this section the simplification procedure in LeakageVerif and highlight its key points, especially regarding equivalence comparison.

Simplification. LeakageVerif implements a lot of simplifications, and an important part of the complexity of the tool lies in them. They include rules like constant propagation, boolean simplifications, factorisation, n-ary operators, or redundancy elimination. A rule of thumb that all simplification rules must respect is that the number of operators after the rule application must be less than before. This guarantees that the size of an expression decreases after a simplification rule is applied, what comes with two benefits: this bounds the number of times a simplification rule can be taken, guaranteeing the convergence; and this helps avoiding that two equivalent expressions do not simplify into the same expression.

The simplification procedure is made in a single pass on the graph, from the leaves up to the root, in such a way that all the opportunities for applying a simplification rule are taken during this pass. This guarantees idempotence, i.e. calling the simplification procedure on an already simplified expression will let it unchanged. It is an important property, as it allows to have a cache for already simplified expressions. Each time the simplification function is called on a node, a link towards the simplified equivalent node is kept in the original node for future calls to this function.

The substitution and simplification rules do not change whether the expression is a bit decomposed expression or not. In addition to a link towards a simplified equivalent expression, each node thus also has a cache for a simplified equivalent node using single-bit variables, and a cache for the extraction of each of its bits.

Finally, the simplification process is designed to be efficient: if the simplification of an expression is a recursive process, going from the leaves up to the root, the simplification of a node whose children have already been simplified only accesses the node and its direct children, and does not need to go all the way down in the expression.

Equivalence Comparison. The simplification process relies on equivalence comparison in many aspects (e.g. when two equivalent expressions are xor'ed, they must be replaced by 0). Therefore, having an efficient comparison is required for having an efficient simplification.

Algorithm 4 Creation of the new graph for a selected mask and the corresponding node to replace

```
Require: node the root of the expression; selectedMask a mask variable; nodeToReplace a masking node in the graph, masked
    with selectedMask
Ensure: All occurrences of nodeToReplace in node have been replaced with selectedMask, and all occurrences of selectedMask
    in node have been replaced with nodeToReplace
procedure GetReplacedGraph(node, selectedMask, nodeToReplace)
    procedure GetReplacedGraphRec(node, selectedMask, nodeToReplace, newNodes)
        if node is not an operator then
            return node
        children $\leftarrow$ []
        for child in node.children do
            if child $\in$ newNodes then
                children.append(newNodes[child])
            else if child is selectedMask then
                children.append(nodeToReplace)
            else if child is nodeToReplace then
                children.append(selectedMask)
            else if selectedMask appears in child then
                newChild $\leftarrow$ GetReplacedGraphRec(child, selectedMask, nodeToReplace, newNodes)
                children.append(newChild)
            else
                children.append(child)
        n $\leftarrow$ Node(node.operator, children)
        newNodes[node] = n
        return n
    return GetReplacedGraphRec(node, selectedMask, nodeToReplace, {})
```

In LeakageVerif, a hash value is computed for each node upon creation. It is applied on the operation of the node itself, and on the hash values of its children. It is safe as the expression represented by a node is guaranteed to remain identical over time. In order to ensure the uniqueness of the hash of a given expression, the children of a commutating operator have to be order in a unique manner: this is achieved via the hash values of the children. The hash function used is SHA-256, which makes the risk of a collision negligible. This hash value allows to compare two expression graphs by simply comparing their hashes in O(1) time.

4 Experimental Evaluation

4.1 Criteria

We compare LeakageVerif to three state-of-the-art available tools designed for verifying masked implementations: SELA, maskVerif and QMVerif. We define hereafter three criteria in order to compare these tools:

- **Expressiveness**. This criterion seeks at defining the scope of uses each tool can cover. In order to make this comparison, we implemented 46 benchmarks and we looked at which benchmarks could be implemented on each tool.
- **Performances and scalability**. This criterion seeks at determining the impact of each tool via the analyses it can complete on its supported benchmarks, depending on the size of the program to verify. In order to make this comparison, we ran all the benchmarks on each possible tool and checked whether it completed within 24 hours. Crash or memory errors are also reported.
- **Accuracy**. All considered tools are sound, in the sense that they cannot miss leakages in expressions, but they are all incomplete without relying on the expensive and non scalable method of exhaustive enumeration. Thus, this criterion aims at comparing the accuracy of the different implementations of verification algorithms only, that is without exhaustive enumeration.

4.2 Benchmarks

There is no simple way to fairly compare state-of-the-art tools. First, there is no standard set of masked implementations. Second, each tool has its own input description language, level of abstraction and input variable natures; in particular, maskVerif directly uses shares to describe the inputs, while the other tools use secret and mask variables. Third, each masked implementation is also designed regarding a specific leakage model and security notion.

Therefore, we decided to build a set of representative benchmarks with a version for each of the considered tool. To this end, we searched the literature to find various examples of masked algorithm descriptions, and then wrote or translated them for each tool, when possible. The resulting set of benchmarks constitutes MaskedVerifBench, which we provide as an open-source collection of masked benchmarks designed for masking verification.

For each selected benchmark, we describe what we used as a starting point for writing the benchmark. We also precise how we derive a version for each tool. We sometimes resorted to so-called *generator*, a program that we made in order to generate the code of a benchmark specifically for one tool. However, as SELA and LeakageVerif have close input formats, only one generator was used for both.

The following 46 benchmarks were used:

- ISW AND: this is the basic ISW scheme applied for the logical AND with two shares [19].
- P1 to P17: these are 17 boolean C++ programs from the publicly available cryptographic software implementations [11]. Among these programs, some are leaky while some others are not. Generators translating the C++ code for each tool were used, as programs P12–P17 are very long and cannot be written manually.

Benchmark	#Verifications	#Leakages	`maskVerif`	`QMVerif`	`SELA`	`LeakageVerif`
ISW AND	8	0	✓	✓	✓	✓
Secmult	355	0	✓	✓	✓	✓
SecmultSM	10	0		✓		✓
A2B01	47	0?		✓	✓	✓
B2A01	8	0		✓	✓	✓
A2B14	182	0		✓	✓	✓
B2A14	364	0?		✓	✓	✓
B2A17	12	0		✓	✓	✓
TI-Add	94	?	✓		✓	✓
TI-Add-Opt	78	?	✓		✓	✓
TI-Add-8	58	?	✓	✓	✓	✓
TI-Add-Opt-8	50	?	✓	✓	✓	✓
Sbox1	51	0		✓		✓
Sbox2	43	0		✓		✓
Sbox3	65	0		✓		✓
Sbox4	69	0		✓		✓
Sbox5	115	0		✓		✓
Sbox6	103	0		✓		✓
Sbox7	167	0		✓		✓
Sbox8	173	0		✓		✓
k^3	11	2		✓		✓
k^{12}	13	2		✓		✓
k^{15}	23	2		✓		✓
k^{240}	25	2		✓		✓
k^{252}	33	2		✓		✓
k^{254}	41	2		✓		✓
AES-Herbst	2666	0		✓	✓	✓
AES-SM	1424	0		✓		✓
AES-FSE13	17602	0		✓	✓	✓
P1	33	16	✓	✓	✓	✓
P2	25	8	✓	✓	✓	✓
P3	7	1	✓	✓	✓	✓
P4	7	0	✓	✓	✓	✓
P5	9	0	✓	✓	✓	✓
P6	10	3	✓	✓	✓	✓
P7	12	2	✓	✓	✓	✓
P8	19	3	✓	✓	✓	✓
P9	19	2	✓	✓	✓	✓
P10	29	2	✓	✓	✓	✓
P11	29	2	✓	✓	✓	✓
P12	196801	0	✓	✓	✓	✓
P13	196801	4800	✓	✓	✓	✓
P14	196801	3200	✓	✓	✓	✓
P15	198401	3200	✓	✓	✓	✓
P16	196801	4800	✓	✓	✓	✓
P17	204801	17600	✓	✓	✓	✓

Table 1

Masked programs used for benchmarking tools, with their number of expressions to verify (`#Verifications`) and the tools supporting them. The number of expressions to verify is the one for `LeakageVerif` and can vary slightly from one tool to another. Column `#Leakages` gives the real number of leaking expressions, based on the results of our experiments and other experiments in the literature.

- `Secmult`: this is the secure Galois field multiplication as described in [28]. The codes for the different tools was written directly by hand, unrolling loops when necessary.
- `SecmultSM`: this is the Secmult algorithm with symbolic multiplication. As `SELA` and `maskVerif` do not support this operation, this benchmark was run only on `QMVerif` and `LeakageVerif`.
- `A2B01` and `B2A01`: these are arithmetic to boolean and boolean to arithmetic masking conversion programs, as presented in [16]. Generators were written for `QMVerif`, `SELA` and `LeakageVerif` based on the pseudo-code given in the article. As arithmetic operations are used, `maskVerif` does not support this benchmark.
- `A2B14` and `B2A14`: these are arithmetic to boolean and boolean to arithmetic masking conversion programs, as designed in [7]. Generators were written from the algorithmic description for `QMVerif`, `SELA` and `LeakageVerif`.
- `B2A17`: this is a boolean to arithmetic masking conversion program, described in [6]. Generators were written from the algorithmic description for `QMVerif`, `SELA` and `LeakageVerif`.
- `TI-Add`, `TI-Add-Opt`, `TI-Add-8` and `TI-Add-Opt-8`: the first two benchmarks are assembly programs performing the masked addition of two 32-bit masked values, given in [20]. The versions ending with `'8'` are 8-bit adaptations of the 32-bit programs: we made those because `QMVerif` only supports 8-bit variables, and cannot implement the 32-bit versions. The programs were translated manually from assembly to `LeakageVerif` programs, then a generator was used to translate them for other tools. All tools can theoretically support these benchmarks, except `QMVerif` for the 32-bit versions.
- `Sbox1` to `Sbox8`: these are different implementations of a masked SBox, based on an idea in [25]. Table 2 gives the details for each version. All versions were written entirely using generators, starting from the algorithms and description given in the referenced articles. Due the presence of symbolic multiplications, these could not be implemented in `maskVerif` and `SELA`.
- `AES-Herbst`: this is the AES with the masking scheme described in [18], comprising the key schedule and the ten rounds. We wrote a generator for `QMVerif` in order to

unroll loops. Besides, we had to overcome the following problem. In this benchmark, the SBox is masked and replaced with an array SBoxP such that `SBoxP[x ⊕ m] = SBox[x] ⊕ m'`, where `m` and `m'` are two masks. The verification can replace atomically the expression `SBoxP[x]` with `SBox[x ⊕ m] ⊕ m'`. However, as `QMVerif` verifies all the results of binary operations, it necessarily verifies the expression `x ⊕ m` before verifying `SBox[x ⊕ m]` , which leaks since `x` is masked by `m`. We used a bug found in `QMVerif` to overcome this problem, which is that the Sbox operation is implemented as the identity (see appendix B). This allows to do the masking with `m'` before the demasking. Finally, the presence of array accesses prevents an implementation on `maskVerif`.

- `AES-SM`: this is a masked implementation adapted from [29]. It implements the same masking scheme as the one in `AES-Herbst`, but with a symbolic Galois field multiplication by constants 2 and 3 in the mix-columns step, and does not mask the key schedule. A generator was used for `QMVerif` as well as the same SBox trick as in `AES-Herbst`, while it is not implementable for `SELA` and `maskVerif`.
- `AES-FSE13`: this is an implementation of the AES following the scheme in [8]. The code was given to us as a `QMVerif` file by the authors of this tool. We used a generator for creating the `LeakageVerif` and `SELA` files. It seemed to us that some lines in the benchmarks were redundant and possibly incorrect (namely `x000 = x000 ^ x000` and `x001 = x001 ^ x001` which we think should be `x000 = x000 ^ k000` and `x001 = x001 ^ k001`), but we chose to not modify the original benchmark file.
- $\texttt{k}^{3}$ to $\texttt{k}^{254}$: following an idea in [25], these programs are buggy implementations of the exponentiation used in the AES. They were obtained via a generator for `QMVerif` and `LeakageVerif`.

	Masking Order		Refresh	Secmult		Power254	
	1^{st}	2^{nd}	[28]	[28]	[5]	[28]	[8]
Sbox1	✓		✓		✓	✓	
Sbox2	✓		✓	✓		✓	
Sbox3	✓			✓			✓
Sbox4	✓				✓		✓
Sbox5		✓	✓		✓	✓	
Sbox6		✓	✓	✓		✓	
Sbox7		✓		✓			✓
Sbox8		✓			✓		✓

Table 2
Specification of the different versions of the SBox used, based on an idea in [25]. Columns *Masking Order* give the masking order (note that only order 1 verification is made in experiments); column *Refresh* indicates that a refreshing mask procedure is used; columns *Secmult* give the version of the secured multiplication used; columns *Power254* give the version for the exponentiation to the value 254.

All benchmarks were run on a server with an Intel CPU Xeon E5-2640v4, with 128GB of memory, under the Scientific Linux 7 operating system. `QMVerif` was compiled with the `-O2` option and always used with the option `-sim` which does some simplifications, as we noticed that it helped the tool a lot for some benchmarks, preventing it from running out of memory, and never had a negative impact[1].

4.3 Expressiveness

We believe that the presented list of algorithms and source code files is a representative set of masked implementations, and covers a quite comprehensive set of what a verification algorithm should be able to do. We summarize in Table 1 the different benchmarks, and which tool is theoretically able to pass them.

We can see in Table 1 that `LeakageVerif` is the only tool which can theoretically support all the benchmarks. We want to add that there is no benchmark that was not included because we could not run it on `LeakageVerif`, and we did not select the benchmarks *a priori* because it could or could not be implemented on some tools. This shows that `LeakageVerif` is adapted to support a wide variety of masked programs, as well high-level algorithmic descriptions as low-level ones. `maskVerif` fails to support benchmarks with arithmetic operations, while `QMVerif` absence of support for variables with a size different from 8 bits prevents it from supporting the `TI-Add` original codes. Also, while `QMVerif` has internal shift operators, but they can never be returned by the parser. Thus, we modified the parser to add it, allowing the tool to support `Secmult`, `TI-Add-8`, `TI-Add-Opt-8`, and `AES-Herbst`. Finally, `SELA` has the addition but no symbolic Galois field multiplication, what prevents it from supporting `SecmultSM`, `Sbox` programs, `k` programs, and some versions of the AES.

For each benchmark, in column `#Leakages`, we give the number of leakages as reported by previous work [25] or the authors of the benchmark. For benchmarks A2B01 and B2A14, we have not been able to prove the absence of leakage : all the tools passing the benchmarks reported possible leakages (cf. section 4.5), but manually analysing the first expressions reported, we found out that they were false positives. However, we have not done this analysis for all the possibly leaking expressions. As `maskVerif` cannot pass these programs and `QMVerif` enumeration could not finish any of these two benchmarks, it is still possible that real leakages exist. The article presenting the `TI-Add` benchmarks does not explicitly state neither the absence of leakage in the code nor the considered leakage model. Our experiments reported possible leakages (cf. section 4.5), and a manual analysis on the first possible leaking expressions revealed that they were indeed leaking. As we did not perform the manual analysis for all instructions, we only know that the number of leakages is comprised between 1 and the values reported by `LeakageVerif`.

4.4 Performances and scalability

We now evaluate the execution time of each benchmarks on the available tools. One problem we had to face, in order to compare the execution, is that `maskVerif` stops at the first leakage encountered, while `QMVerif` verifies the whole program, and it is a user decision for `SELA` and `LeakageVerif`. In order to compare the three tools, we decided to modify `QMVerif` to make it stop after the first leakage encountered, and used this option as well for `SELA` and `LeakageVerif`. Table 3 thus presents the execution times up to the first leakage detected. For a fair comparison, we verified manually that for all cases where a possible leakage is detected in a benchmark, it is the same leakage for all tools.

[1] This option was not used for `AES-Herbst` because it provokes an error

Benchmark	#instructions analysed	maskVerif	QMVerif	SELA	LeakageVerif
ISW AND	8	< 0.1s	0.2s	< 0.1s	0.2s
Secmult	355	< 0.1s	*OoM*	2.7s	0.7s
SecmultSM	10		0.4s		< 0.1s
A2B01	8		*Timeout*	**< 0.1s**	**< 0.1s**
B2A01	8		17s	< 0.1s	< 0.1s
A2B14	183		1.7s	5.6s	0.9s
B2A14	364		*OoM*	*OoM*	**10s**
B2A17	12		1m 30s	1.6s	0.2s
TI-Add	11	**< 0.1s**		*OoM*	**0.3s**
TI-Add-Opt	7	**< 0.1s**		**0.1s**	**0.1s**
TI-Add-8	11	**< 0.1s**	*OoM*	**< 0.1s**	**< 0.1s**
TI-Add-Opt-8	7	**< 0.1s**	*Timeout*	**< 0.1s**	**< 0.1s**
Sbox1	51		0.2s		< 0.1s
Sbox2	43		0.2s		< 0.1s
Sbox3	65		0.7s		0.1s
Sbox4	69		0.8s		0.1s
Sbox5	115		0.3s		0.2s
Sbox6	103		0.2s		0.2s
Sbox7	167		1.2s		0.2s
Sbox8	173		1.2s		0.2s
k^{3}	4		**4m 8s**		**< 0.1s**
k^{12}	4		**4m 1s**		**< 0.1s**
k^{15}	4		**4m 3s**		**< 0.1s**
k^{240}	4		**4m 16s**		**< 0.1s**
k^{252}	4		**4m 14s**		**< 0.1s**
k^{254}	4		**4m 6s**		**< 0.1s**
AES-Herbst	2666		*assert fails*	*OoM*	20s
AES-SM	1425		*OoM*		0.5s
AES-FSE13	17602		*OoM*	*OoM*	18s
P1	1	**< 0.1s**	**< 0.1s**	**< 0.1s**	**0.1s**
P2	9	**< 0.1s**	**< 0.1s**	**< 0.1s**	**< 0.1s**
P3	7	**< 0.1s**	**0.2s**	**< 0.1s**	**< 0.1s**
P4	7	< 0.1s	0.5s	< 0.1s	< 0.1s
P5	9	< 0.1s	0.5s	< 0.1s	< 0.1s
P6	8	**< 0.1s**	**55s**	**< 0.1s**	**< 0.1s**
P7	11	**< 0.1s**	**0.5s**	**< 0.1s**	**< 0.1s**
P8	13	**< 0.1s**	**53s**	**< 0.1s**	**< 0.1s**
P9	13	**< 0.1s**	**56s**	**< 0.1s**	**< 0.1s**
P10	25	**< 0.1s**	**29m 12s**	**< 0.1s**	**< 0.1s**
P11	24	**< 0.1s**	**26m 35s**	**< 0.1s**	**< 0.1s**
P12	196801	*OoM*	5.8s	3m 8s	17m 19s
P13	1672	**4m 20s**	**5.0s**	**6.3s**	**5.8s**
P14	1711	**4m 24s**	**1m 7s**	**6.5s**	**5.8s**
P15	1717	**4m 40s**	**25m 5s**	**6.0s**	**6.2s**
P16	1712	**4m 17s**	**4.9s**	**6.4s**	**6.2s**
P17	1712	**5m 30s**	**5.1s**	**6.3s**	**6.5s**

Table 3

Execution times for every benchmarks for all tools up to the first detected leakage. The second column gives the number of instructions analysed up to the first detected leakage. A blank cell indicates that the tool could not run the benchmark. Times in bold indicate that a leakage was found. *Timeout* indicates that the verification did not finish in 24 hours. *OoM* indicates that the verification ran out of memory.

Looking at these results, we can see that, maskVerif performs very well on small programs, but tends to have significantly longer execution times for longer programs, namely P13 to P17. It is relevant to note that for these programs, the time taken to read the input file is non-negligible: around 5 seconds for LeakageVerif for example, which detects a leakage immediately after. QMVerif runs out of memory on 5 benchmarks, has two *timeouts*, and one assert failure. Besides, it is significantly slower than LeakageVerif on 15 benchmarks (B2A01, B2A17, k^{3} to k^{254}, P6, P8 to P11, P14 and P15). It is significantly faster than LeakageVerif only on P12: we tried to investigate the reason of this good performance compared to the other tools (QMVerif and SELA run out of memory while LeakageVerif passes almost all of its time on one instruction near the end), but so far we could not isolate the rule allowing for this quick conclusion. Finally, SELA runs out of memory in 4 benchmarks due to its lack of scalability. It does not manage to complete the analysis of the first eleven instructions of TI-Add, and cannot pass more than 2 rounds of the AES-Herbst. Overall, we can conclude that LeakageVerif has a clear global advantage over the other tools.

In order to have a more in-depth comparison, we let QMVerif, SELA and LeakageVerif complete their analysis on the programs for which a leakage was detected. Indeed, it is often more interesting to get all the leaking expressions in an program in a single verification than having to correct the leaking parts one by one and rerun a verification after each correction. It is also an interesting use-case as it stresses more substitution methods, since a leaking expression will often select and make replacements with all masks before failing to conclude. If not handled correctly, the expression size can grow a lot and limit the tool scalability. Table 4 presents the verification time of the whole program, for programs with at least one leakage, along with the number of leaking expressions detected for each tool.

We can notice that when the analysis completes, the three tools detect the same number of leakages. We can also notice that for all the k^{i} programs, QMVerif takes around one hundred minutes, while LeakageVerif takes less than 0.1s. For programs P1 to P11, QMVerif does three quick veri-

Benchmark	QMVerif		SELA		LeakageVerif	
	Leaks	Execution time	Leaks	Execution time	Leaks	Execution time
A2B01		*Timeout*	35	10m 4s	35	1.9s
B2A14		*OoM*		*OoM*	1	10s
TI-Add				*OoM*	70	5m 7s
TI-Add-Opt				*OoM*	66	1m 56s
TI-Add-8		*OoM*		*OoM*	36	2.2s
TI-Add-Opt-8		*Timeout*		*OoM*	38	1.4s
k^3	2	99m 29s			2	< 0.1s
k^{12}	2	104m 2s			2	< 0.1s
k^{15}	2	106m 10s			2	< 0.1s
k^{240}	2	108m 55s			2	< 0.1s
k^{252}	2	106m 2s			2	< 0.1s
k^{254}	2	109m 59s			2	< 0.1s
P1	16	< 0.1s	16	< 0.1s	16	< 0.1s
P2	8	0.3s	8	< 0.1s	8	< 0.1s
P3	1	< 0.1s	1	< 0.1s	1	< 0.1s
P6	3	2m 10s	3	< 0.1s	3	< 0.1s
P7	2	1m 8s	2	< 0.1s	2	< 0.1s
P8	3	3m 28s	3	< 0.1s	3	< 0.1s
P9	2	2m 2s	2	< 0.1s	2	< 0.1s
P10		*OoM*	2	< 0.1s	2	< 0.1s
P11		*OoM*	2	< 0.1s	2	< 0.1s
P13	4800	84m 12s	4800	3m 8s	4800	13m 57s
P14	3200	4116m 20s	3200	3m 9s	3200	13m 54s
P15		*OoM*	3200	3m 34s	3200	8m 0s
P16		*Seg. Fault*	4800	3m 11s	4800	13m 58s
P17	17600	29m 30s	17600	3m 15s	17600	13m 31s

Table 4
Execution times for the entire analysis of programs with leakages for QMVerif, SELA and LeakageVerif. *OoM* indicates that the verification ran out of memory.

fications, four verifications between 1 and 3 minutes, while two verifications run out of memory. SELA and LeakageVerif on the other hand, complete all the analysis in less than 0.1s. For longer programs P13 to P17, QMVerif does two *Out of Memory* and is significantly slower than SELA and LeakageVerif for the three other benchmarks. We can also notice that SELA is faster than LeakageVerif: this is due to the fact that the expression graphs of these programs are not folded (in the sense that an intermediate computation result is never used twice), combined to the fact that there are a big number of variables. Therefore, maintaining meta-information in the nodes of the expression is very costly for LeakageVerif when substitutions occur by the end of the program, while it does not bring interest in this particular case. However, we consider this overhead as acceptable as it does not compromise scalability. On the other hand, SELA runs out of memory on all TI-Add benchmarks, while LeakageVerif is the only tool able to complete them.

Finally, SELA's lack of scalability can also be seen on A2B01, where it takes more than 10 minutes to complete, whereas LeakageVerif takes less than 2 seconds. In order to reach maximum performance, maskVerif and LeakageVerif could probably benefit from being written in a lower level language instead of a high-level one, and maybe gain up to an order of magnitude in time. Yet, the most important aspect is scalability with the growth of the expression, for which this cost does not make much difference in the end. Added to that, the development in a lower-level language would be much longer and subject to a lot more bugs.

Regarding memory exhaustion, if QMVerif and SELA memory consumption quickly grows and several benchmarks exhaust the 128GB limit, memory consumption remain quite low for LeakageVerif since the memory used is below 1GB, except for P12-P17 where it is around 8GB.

Finally, for QMVerif, we noticed that the fast verifications – the one during less than a minute – were the ones for which the enumeration was not called. This raises the question of the accuracy of the inference alone, compared to the substitution approach implemented in other tools.

4.5 Accuracy

As mentioned above, both verification methods by inference and by substitution are sound, but the tools can claim to be complete only by resorting to a costly exhaustive enumerative approach when they cannot conclude with their inference or substitution rules only. In order to compare the accuracy of the inference rules in QMVerif with the substitution approach in LeakageVerif, we deactivated the enumeration in QMVerif, counting instead a potentially false positive when the inference failed to conclude. We did not do the same for maskVerif due to our lack of understanding Caml code, and thus our inability to do the same modification.

Table 5 presents the number of false positives for QMVerif and LeakageVerif without enumeration, i.e. the number of expressions for which a tool cannot conclude threshold probing security, while the expression does not leak (SELA is not included since it has the same number of leakages as LeakageVerif for the benchmarks it can complete). We can observe that for the two benchmarks LeakageVerif has at least a false positive (namely A2B01 and B2A14), QMVerif has more false positives (37 vs. 35 and 100 vs. 1), while it also has false positives for 9 other benchmarks. QMVerif found 64 possible leakages for TI-Add-8 and TI-Add-Opt-8. However, LeakageVerif reported only 36 and 38 possible leakages: therefore, QMVerif has at least 28 and 26 false positives respectively for these benchmarks. Last but not least, even with enumeration deactivated, QMVerif does not manage to complete the analysis of any of the AES benchmarks.

Benchmark	`QMVerif`	`LeakageVerif`
ISW AND	0	0
Secmult	1*	0
SecmultSM	0	0
A2B01	37	35
B2A01	1	0
A2B14	0	0
B2A14	100	1
B2A17	1	0
TI-Add		0?
TI-Add-Opt		0?
TI-Add-8	> 28*	0?
TI-Add-Opt-8	> 26*	0?
k^3	0	0
k^{12}	0	0
k^{15}	0	0
k^{240}	0	0
k^{252}	0	0
k^{254}	0	0
Sbox1	0	0
Sbox2	0	0
Sbox3	0	0
Sbox4	0	0
Sbox5	0	0
Sbox6	0	0
Sbox7	0	0
Sbox8	0	0
AES-Herbst	*Timeout*	0
AES-SM	*Timeout*	0
AES-FSE13	*OoM*	0
P1	0	0
P2	0	0
P3	0	0
P4	3	0
P5	2	0
P6	0	0
P7	2	0
P8	0	0
P9	0	0
P10	0	0
P11	3	0
P12	0	0
P13	0	0
P14	0	0
P15	0	0
P16	0	0
P17	0	0

Table 5
Number of false positives found by `QMVerif` with enumeration disabled and `LeakageVerif`. *: the `-sim` option of `QMVerif` was deactivated because it provoked an error during a z3 simplification.

Finally, we do not know if the time taken by `maskVerif` for `P13` to `P17` to reach the first leakage is due to enumeration because their substitution algorithm is less precise than the one implemented in `LeakageVerif`, or for another reason.

5 Conclusion and Future Work

We presented a efficient and scalable approach, based on a substitution method, for verifying the absence of first order leakage in symbolic expressions, using a substitution approach. This approach was implemented in a tool, `LeakageVerif`, which we compared to three state-of-the-art other tools on 46 benchmarks, and showed that `LeakageVerif` performs better than these tools in terms of expressiveness regarding supported programs, scalability regarding execution time and memory used, and accuracy of its results when not relying on enumeration. The tool as well as the benchmarks written for all evaluated tools will be released in the hope that they will be useful to the community and for reproducibility.

Future work includes the support in `LeakageVerif` of hardware circuits with glitches and registers, as well as supporting stronger security notions and higher verification orders.

References

[1] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *European Symposium on Research in Computer Security*, pages 300–318. Springer, 2019.

[2] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer, 2015.

[3] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. maskverif: automated analysis of software and hardware higher-order masked implementations. Technical report, Technical Report 562, 2018.

[4] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, pages 146–166. Springer, 2014.

[5] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 441–458. Springer, 2014.

[6] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 93–114. Springer, 2017.

[7] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 188–205. Springer, 2014.

[8] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In *International Workshop on Fast Software Encryption*, pages 410–424. Springer, 2013.

[9] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *6th International Workshop on Security Proofs for Embedded Systems (PROOFS)*, 2017.

[10] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Side-channel robustness analysis of masked assembly codes using a symbolic approach. *Journal of Cryptographic Engineering*, 9(3):231–242, 2019.

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

[12] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. Qms: Evaluating the side-channel resistance of masked software from source code. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[13] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. Quantitative masking strength: quantifying the power side-channel resistance of software code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1558–1568, 2015.

[14] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. Quantitative verification of masked arithmetic programs against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 155–173. Springer, 2019.

[15] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(3):1–32, 2019.

[16] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer, 2001.

[17] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.

[18] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An aes smart card implementation resistant to power analysis attacks. In *ACNS*, volume 3989, pages 239–252. Springer, 2006.

[19] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[20] Bernhard Jungk, Richard Petri, and Marc Stöttinger. Efficient side-channel protections of arx ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 627–653, 2018.

[21] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[22] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[23] Quentin L. Meunier, Inès Ben El Ouahma, and Karine Heydemann. Sela: a symbolic expression leakage analyzer. In *International Workshop on Security Proofs for Embedded Systems*, 2020.

[24] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[25] Gao Pengfei, Xie Hongyi, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. Formal verification of masking countermeasures for arithmetic programs. *IEEE Transactions on Software Engineering*, 2020.

[26] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.

[27] Microsoft Research. Z3py-python interface for the z3 theorem prover, 2012.

[28] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.

[29] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 57–64. IEEE, 2018.

[30] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. Sc infer: refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*, pages 157–177. Springer, 2018.

Appendix A
Bug found in maskVerif

Let us consider the program in Figure 9, which contains two secret 8-bit inputs a and b split respectively in (a[0]), a[1]) and (b[0], b[1]). The expression computed on the 4 most significant bits is equal to the four least significant bit of a. It is a clear value leakage, however, verifying threshold probing security for this expression leads maskVerif to conclude that there is no leakage.

```
proc test_shift:
  inputs: w8 a[0:1], w8 b[0:1]
  outputs: w8 c[0];

  c[0] = {( (a[0] << 4) ^w8 (b[0] >> 4) ) ^w8 (a
      [1] << 4)};
end
```

Figure 9. MaskVerif program exhibiting a bug in the tool

Appendix B
Bug found in QMVerif

Let us consider the program in Figure 10, in which a is a secret variable and r a random. The variable e is well masked, and tmp is simply the sbox operator applied to the expression a $\oplus$ r. Since QMVerif considers that the sbox operator is the identity, the variable zero is equal to 0 as there exist a rule for canceling the exclusive-or of two identical expressions. Therefore, the expression of out is computed as 0, since there is a rule to simplify the bitwise and of any value with 0 to 0. As a consequence, no leakage is found, whereas in reality the variable zero is not equal to 0, and some of the bits of out do leak.

```
import require Byte.

module M {
    proc main(a:byte): byte {
        var r, e, out, tmp, zero;
        r = $distr;
        e = a ^ r0;
        tmp = sbox e;
        zero = e ^ tmp;
        out = a & zero;
        return out;
    }
}
```

Figure 10. QMVerif program exhibiting a bug in the tool