SELA: a Symbolic Expression Leakage Analyzer

Quentin L. Meunier, Inès Ben El Ouahma, Karine Heydemann

SELA: a Symbolic Expression Leakage Analyzer

Quentin L. Meunier[1], Inès Ben El Ouahma[1], Karine Heydemann[1]

Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France
{quentin.meunier,ines.ben-el-ouahma,karine.heydemann}@lip6.fr

Abstract

Side-channel attacks are a powerful class of attacks targeting cryptographic devices, which exploit physical quantities during the execution of an algorithm in order to recover key material. Masking is a popular counter measure to mitigate such attacks, and consists in splitting a secret into n shares, such that any combination of $n-1$ shares or less is statistically independent from the secret.

If general masking schemes exist in some cases, masked algorithms are often specific and are required to be verified independently, as the conversion from an unmasked algorithm is not trivial. Existing tools for verifying masked algorithms either target the algorithmic level using abstract constructions or hardware descriptions, but show a lack for a generic tool which can act as a basis for different verification contexts.

In this article, we present SELA, an open-source tool for verifying masked expressions, provided as a python library. We detail and motivate the design choices made in the implementation, in particular regarding the simplification rules and strategy used by the verification algorithm. The internal representation is also compatible with several leakage models. We show the interest of SELA by verifying existing use-cases from different contexts: masked hardware circuits, algorithms and generated assembly code. Besides showing the versatility of the approach, these use-cases also demonstrate the good accuracy of the verification and the efficiency of the implementation.

1 Introduction

Side-channel attacks constitute a very powerful class of attacks targeting cryptographic devices. They aim at recovering key material by the means of recording and analyzing physical quantities during the execution of the algorithm. Many successful side-channel attacks have been discovered over the past two decades, many of them relying on the analysis of power consumption [18, 6, 7, 19, 12, 17]. Resistance against such attacks has thus become a major concern, and a lot of research work has been done recently in order to assess such leakages [26, 8, 23].

Masking is a popular countermeasure against such attacks. Masking at order n consists in splitting a secret, or sensitive data, into $n+1$ shares, in such a way that any combination of n shares or less does not have a statistical dependency with the secret. Masking is usually boolean, meaning that all the shares need to be xor-ed in order to get the secret, although arithmetic masking also exists [13]. The higher the n value, the better the resistance is against attacks. However, high-order masking comes at the price of an increased resulting complexity of the implementation. Moreover, transforming an algorithm or circuit in a masked equivalent is

not trivial, and although some general schemes have been proposed for circuits [16, 21, 22, 14], it often requires an *ad-hoc* scheme. The higher the order of sharing, the more costly the transformation becomes. Besides, sharing at order n does not imply a resistance at order n: the resistance order inherently models the capabilities of an attacker. Put simply, resistance at order d is achieved when an attacker that can have access to d internal values in the circuit or algorithm cannot learn any secret information. The resistance order is different from the sharing order because the observation of one value could give him information on several shares. For these reasons, there is a need to assess the achieved security of a masked implementation.

As masking is a provable countermeasure [25], verification algorithms and tools have been proposed to help assess the security of software or hardware implementations of masked algorithms [3, 9, 28]. Existing verification tools target a specific abstraction level or implementation type, e.g. algorithmic level with abstract constructions, or hardware description language. The verification algorithms work on trees or graphs representing masked expressions and are based either on inference rules or a substitution approach to prove the absence of leakage. However, the published algorithms often hide details which can be of high importance for the performance and accuracy of the implemented analysis. In particular, verification algorithms often need to simplify expressions in order to be able to conclude or apply a transformation, but how or when to apply such simplifications is not necessarily fully reported. Also, the leakage model, which is critical to security, is often implicitly specified. Last but not least, the existing tools are often either not available or not open. In order to help face the challenge of masking verification, we believe that there is a need for an open framework, that acts on plain symbolic expressions while being general enough so that it can serve as a basis for different implementation types and abstraction levels. In this paper, we provide such a framework. We also explain the different choices regarding internal data structures and algorithms, and justify our choice using experimental validation.

We believe that this paper makes 3 contributions:

- It provides an easy-to-use python open-source library implementation of an algorithm proving statistical independence of an expression w.r.t. secret variables, along with five leakage models[1];
- It details and motivates the design choices of the implementation of the algorithm;
- It shows the interest of the approach on various examples from the literature, in terms of performance and ease of use.

The rest of the article is organized as follows: Section 2 gives some background on masking schemes and existing tools for verifying masked expressions; Section 3 presents our tool SELA in terms of usage and supported operations, internal representation and algorithms. It also presents the different simplification strategies and the leakage models considered. Section 4 describes the experiments made and the results obtained in terms of performance and accuracy for the proposed simplification strategies. Finally, Section 5 concludes this paper.

2 Background and Related Works

2.1 Masking Schemes

The first scheme for masking a AND gate at order one, called Trichina AND gate, was proposed by [27] in the process of masking the AES. It comes in two versions: one which actually leaks,

[1]The source code is available at `https://www-soc.lip6.fr/~meunier/soft/sela.tar`

and one which requires an additional random value during the computation (as opposed to the random values being required for performing the sharing). Such additional random values are frequent in masking schemes, and are often called *refresh* or *mask*, even if the term *mask* can also refer to initial random values used for the sharing. Besides, this scheme suffers from glitches, i.e. temporary values on wires due to the different propagation times of a gate's inputs.

In the mean time, the first general hardware masking scheme was introduced in [16], and consists in decomposing a circuit in AND, NOT and XOR gates, while providing a d-order resistance implementation for each of these gates. This scheme, called ISW, however did not consider glitches in a circuit either. A third scheme, called Threshold-Implementation (TI), was proposed in [21] for boolean functions (and not only AND gates). It removes the glitch problem and the need for internal random values, but has the drawback to require many more shares and gates for achieving a given level of security. Almost a decade later, [22] presents how the previous schemes are related, and based on the characteristics of these schemes, 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 refresh, i.e. more randomness), and a gain in security for higher orders. Finally, [14] presents Domain-Oriented Masking (DOM), which uses a similar structure to GMS, but achieves an identical level of security with a reduced sharing order, namely order d resistance with d shares for the AND function. This is done by inserting registers inside the computation in order to stop glitch propagation.

If using a general masking scheme is generally well adapted for a circuit, algorithms – i.e. software implementations – often require an ad-hoc scheme, e.g. [15, 1]. Such implementations have no intrinsic guaranties and need to be verified each in a independent manner. Furthermore, if an algorithm can be proven leakage-free, an implementation, using usually lower-level construction, can be leaky. This shows the need for proving algorithms and their implementation at different levels.

2.2 Masking Verification

Verifying a masking scheme can be done naively by enumerating all the combinations of the different variables and checking that an expression has the same distribution w.r.t. the secrets it contains. This approach, however, quickly shows some limits regarding scalability, either in terms of number of variables, or variable width.

On the contrary, symbolic methods do not enumerate variable values but reason on variable types to deduce information on expressions. Several approaches have been proposed for verifying the security of a masked implementation.

A first approach, [3] proposed an algorithm for verifying the absence of leakage of an expression. The property verified, called Non-Interference (NI), states that the joint distribution of a set of expressions is independent from the secrets values contained in the expression. The algorithm is based on the fact that a sub-expression $\texttt{m} \oplus \texttt{e}$ in which `m` is a mask (uniform random value) can be replaced by `m` in the expression if `m` does not appear anywhere else in the expression. This algorithm was first implemented in the EasyCrypt tool [5], but not made accessible. This algorithm was then implemented for the verification of hardware circuit in a tool called maskVerif [2], which takes as input an annotated verilog file containing the circuit to verify. The verification algorithm inside maskVerif was later improved in [4], which notably removed the condition that `m` should not appear anywhere in the expression by a weaker condition. This improved algorithm constitutes one of the basis of this work.

Another approach, [9] introduced a symbolic analysis based on inference. Inference permits

```
m = symbol('m', 'M', 4) # 4-bit variable named 'm' and of type Mask
k = symbol('k', 'S', 4) # 4-bit variable named 'k' and of type Secret
e = (m ^ k) & m         # expression computation
res = checkNIVal(e)     # check for leakage in the expression value
```

Figure 1: Simple example of SELA program

reusing the results of sub-expressions for determining the result of the current expression, while the previous approaches must start from zero at each new expression, even if it is a combination of already analyzed expressions. The advantage of this property is patent since algorithms, like circuits, can be decomposed as a succession of operations re-utilizing the results of previous operations. On the other hand, the inference rules may not be able to conclude in some cases in which the algorithm of [4] could. Besides, the inference approach is currently limited to first-order resistance analysis. 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 [28], then improved in the QMSInfer tool [11]. Finally, support for arithmetic operators was added in the QMVerif tool in [10]. However, none of these tools was made available. Although inference-based algorithms have not been considered in this work, combining both non-inference based and inference based approaches is clearly an axis for future work.

Both symbolic approaches, inference-based or not, are incomplete, although none of them can give false negatives, i.e. no leakage can be missed. However, they can both be completed with a symbolic approach having an opposite strategy, giving false negatives only, and aiming at guaranteeing the presence of leakages in some cases. Finally, these approaches can be combined with an enumerative approach, as in [10] for cases in which they cannot conclude.

To the best of the authors' knowledge, there does not exist an open, easily usable tool for verifying the absence of leakage in a masked expression. This is what SELA aims at providing.

3 SELA

3.1 Overall Presentation

SELA is a python library for checking the absence of leakage of symbolic masked expressions. We chose python due to its wide usage as well as its its ease-of-use. SELA provides a large set of constructions for creating symbolic expressions with symbolic variables, constants and operations on expressions. A symbolic variable has no concrete value and represents a set of values. In SELA, variables must have a type among the three following ones: secret variable, mask variable, or public variable. Besides, all variables and constants must have a specified size expressed as a bit width. The library then permits checking the NI property of created expressions – i.e. statistical independence of the expression w.r.t. the secrets – by providing an implementation of the algorithm described in [4] and different leakage models to be considered for the verification.

Figure 1 shows a simple example of expression construction comprising two occurrences of a mask and one occurrence of a secret, along with the check of the NI property in the value leakage model.

The verification algorithm iteratively replaces masked sub-expressions with the mask itself, until there are no more occurrences of secrets in the expression. Algorithm 1 recalls the NI verification algorithm from [4]. In this algorithm, the simplification of sub-expressions occurs at most once (Alg. 1 Lines 11-14) for performance reason as mentioned by the authors [4].

Algorithm 1 Non-Interference Algorithm, described in [4]

Require: V = (v_1, ..., v_n) n symbolic expressions
Ensure: False is returned if all expressions do not satisfy the NI property; otherwise, True is probably returned

```
1:  masksTaken ← ∅
2:  alreadySimplified ← false
3:  while True do
4:      if there is no secret occurrence in any expression in V then
5:          return True
6:      Select a mask r such that r ∉ masksTaken, and such that there exists a sub-expression w in a v_i bijective w.r.t. r
7:      if r ≠ ∅ then
8:          for all i ∈ [1; n] do
9:              Replace r with w in v_i
10:         masksTaken ← masksTaken ∪ {r}
11:     else if alreadySimplified = false then
12:         alreadySimplified ← true
13:         for all i ∈ [1; n] do
14:             Simplify v_i
15:     else
16:         return False
```

Moreover, no detail is given about the simplification algorithm used. In the opposite case, in Section 3.5, we present our simplification rules and the different simplification strategies that we experimentally evaluate in Section 3.5. The ability of this algorithm to conclude is also related to the selection of the mask and associated bijective sub-expression (Alg. 1 Line 6). The authors in [4] state that they choose the mask by increasing multiplicative depth in the expression tree. In SELA, two parameters are considered for the selection: the number of occurrences of the mask and its depth w.r.t. the root of the expression graph. Algorithm 2 describes how the selection operates in SELA. The function returns two nodes in the expression graph: a mask node (`maskNode`) and an operation node bijective w.r.t the mask node (`bijectNode`). These nodes are selected in order to minimize first the number of occurrences of `maskNode`, and then the depth of `bijectNode`, so as to minimize the number of replacements and the expression complexity.

We can notice that the selection of a mask candidate for a replacement (line 8 of Algorithm 2) requires us to be able to determine the mask occurrences. These occurrences are also needed when a replacement of a sub-expression masked with m is performed since other potential occurrences of m must also be replaced with the original sub-expression (line 10 of Algorithm 1). In both cases, it would be prohibitive to go through the entire expression searching for mask occurrences. Hence, an efficient implementation of the algorithm for checking non-interference requires to operate on a representation of the expression in which it is possible to access to all parents of a leaf. These considerations motivated the design choices made regarding the internal representation used in SELA, as explained in the next section.

3.2 Internal Representation of Expressions

z3 is a SAT solver which can be used, along with its front-end z3py [24], for manipulating symbolic expressions. It handles large expressions particularly well, which makes it a good candidate for our intended goal. Unfortunately, it is not possible to access the parent of a

Algorithm 2 SELA selection algorithm

```
Inputs: M the set of masks; R the set of masks already taken
Returns: a mask node maskNode, and bijectNode, a sub-expression bijective w.r.t. maskNode.
nbMaskOccurrences ← ∅
for m ∈ M such that m ∉ R do
    nbMaskOccurrences[len(m.parents)].add(m)
possibleOpNodes ← map()
for m ∈ M such that m ∉ R do
    possibleOpNodes[m] ← map()
    for parent ∈ m.parents such that parent.op is a bijective operator do
        parentKO ← False
        for p ∈ m.parents \{parent} do:
            if p is in a sub-expression of parent then
                parentKO ← True
                break
        if !parentKO then
            depth ← computeDepth(parent)  ▷ depth from parent up the the root of the expression
            if len(possibleOpNodes[m]) = 0 or depth < possibleOpNodes[m]['depth'] then
                possibleOpNodes[m] ← (parent, depth)
bijectNode ← ∅
for nbOcc ∈ nbMaskOccurrences.keys().sort() do
    for m ∈ nbMaskOccurrences[nbOcc] do
        if len(possibleOpNodes[m]) ≠ 0 and possibleOpNodes[m]['depth'] < minDepth then
            bijectNode, minDepth ← possibleOpNodes[m]
            maskNode ← m
    if bijectNode ≠ ∅ then
        break
return maskNode, bijectNode
```

node in a z3 expression, and as a result, we excluded this possibility. Instead, we used our own representation for implementing graphs modeling expressions on which the Algorithms 1 and 2 are executed, using python classes. Note that graphs modeling expressions are not directly manipulated by the user. Users only manipulate objects of a class which internally gathers two graphs for the current expression: one modeling the expression at the word level (the word graph), and one modeling the expression at the bit level (the bit graph). Both graphs are built in parallel for each operator encountered. The interest of having a word graph is that some expressions can be proven leakage-free independently from their size, which results in a faster verification process as the number of nodes is reduced. However, when this verification fails, SELA can take advantage of the more precise bit graph to verify it and conclude in more cases. Figure 2 shows the two graphs associated to the example expression in Figure 1.

The internal implementation of SELA expressions returns a new expression every time an operator is used, which requires us to copy the operands of the operator. For large expressions, this technique makes the time spent for copying non negligible. For a given set of expressions, the memory footprint of a SELA internal expression is also greater than the one required for a z3py equivalent expression. For limiting memory footprint and reduce the time spent in copying expressions, using z3py for internal expression representation is also supported in SELA: the user can configure which internal representation he wants to use via a configuration file. This choice is transparently handled by SELA : the same code can be run using one representation or the other (except for array accesses). In the following, these internal representations will be

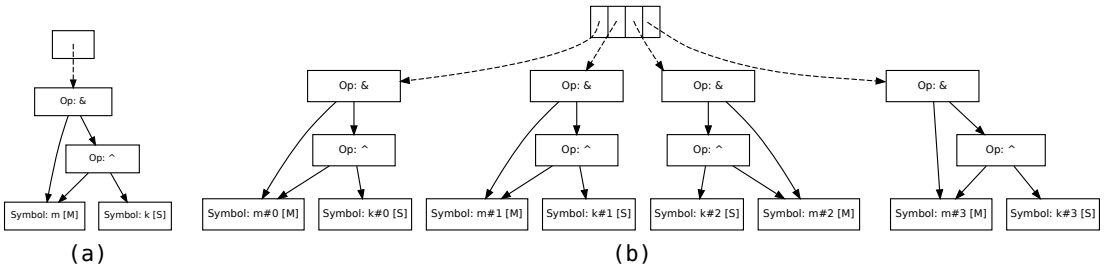


Figure 2: **Word graph (a) and bit graph (a) associated to the expression** `(m ^ k) & m`**.** The bit graph contains four roots corresponding to the four bits of the expression. `s#b` designates bit `b` of symbol `s`, while [S] and [M] designate respectively secret and mask nodes.

referred to as SELA Ex and SELA z3.

It is important to note that even in the case of using SELA z3, the expression is still translated in the form of a graph in the native SELA representation but only once before launching the verification algorithm on it. If having two possible expression representations is not ideal, results in the experimental section show that SELA Ex is more efficient on small programs and concludes on more cases with some strategies, while SELA z3 is more scalable.

3.3 Supported Operations

In order to make SELA programs work easily with both types of underlying representation, operators names and parameters order are those of z3py. Currently, the following operations are supported:

- `&`, `|`, `^`, `~`: bitwise logical AND, OR, XOR and NOT
- `+`, `-`: arithmetic addition and subtraction
- `>>`: arithmetical shift right
- `<<`: logical shift left
- `LShR(e, n)`: logical shift right of expression `e` of `n` bits
- `Extract(msb, lsb, e)`: extraction in expression `e` of the bits starting at index `lsb` up to index `msb` included (`msb`, `lsb`: naturals, `msb` $\geq$ `lsb`)
- `Concat(e, f)`: concatenation of expressions `e` and `f`
- `ZeroExt(nz, e)`: concatenation of `nz` zeros at the left of expression `e`
- `SignExt(nz, e)`: concatenation of the most significant bit (sign bit) of `e` `nz` times at the left of expression `e`

3.4 Leakage Models

Verifying the statistical independence of an expression to a secret variable makes the assumption that the value of the expression is leaking. This corresponds to the first order probing security property, as defined in [4]. However, it is often not sufficient: indeed, at a low-level, power consumption is mainly affected by the change of state of transistors [20]. Therefore, transitions between expressions are highly relevant to the leakage of a system, if they can be known, whether between variables or registers.

To this end, SELA implements the verification for five leakage models, a value-based one and four transition-based ones. They are described as follows, along with the implementation technique used for performing the corresponding verification using Algorithm 1.

- **Value.** This leakage model considers the value of a single expression. The expression is passed directly to Algorithm 1.
- **Transition.** This leakage model considers the most general notion of transition. Informally, it considers that the two expressions which compose the transition can leak any operation combining these expressions. As such, it reasons on the product of the distribution of the two expressions. It is implemented by verifying the set made of the two expressions with the NI algorithm. Since this leakage model is very general, it may not translate into real leakages, while it is harder to make a system secure w.r.t. this model. This motivated the definition of the following weaker models.
- **TransXor.** As in hardware the power consumption is highly related to bits changing of state, a meaningful power consumption model in hardware consists in the exclusive-or, or "xor", of two consecutive expressions appearing in a hardware component. The verification in this leakage model thus considers two expressions that the user thinks are consecutive in hardware, and calls Algorithm 1 on the resulting xor.
- **TransBit.** This leakage model is similar to the `Transition` model, but for each bit individually. A leakage found in this model should result in a more easily observable physical leakage. The verification is made by calling Algorithm 1 n times for two n-bit expressions, each time on a set of two single-bit expressions.
- **TransXorBit.** This leakage model is a combination of the `TransBit` and `TransXor` models. The verification is carried out independently on each bit of the expression resulting from the xor of two consecutive expressions.

Figure 3 shows how these different leakage models are linked together, while Figure 4 shows various examples of 2-bit expressions and their resistance to the five leakage models presented.

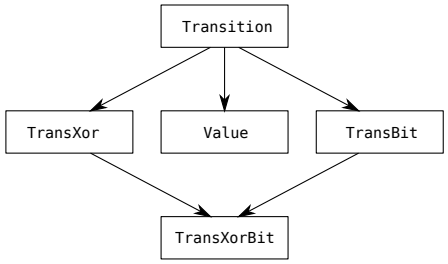
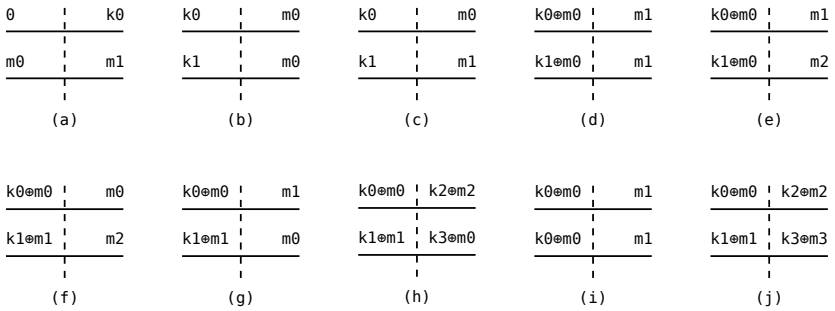


Figure 3: **Graph of relations of the different leakage models implemented in SELA.** A model at the origin of an arrow is stronger than the model at the end of the arrow. No leakage in a given model implies no leakage in the weaker models. A leakage in a given model implies a leakage in the stronger models.

3.5 Simplifications Rules and Strategies

SELA implements a `simplify()` function on graphs which is applied starting from the leaves, and up to the root. This section describes the main simplifications implemented by this function. Constant propagation is made whenever possible, as long as merging a node with its children for associative operators. Although we do not formally describe the simplification algorithm, Table 1 presents the simplification rules.

The description of Algorithm 1 makes only one simplification of the expression: in case of a failure, the expression is simplified and the verification algorithm is then re-executed.



Leakage Model	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
Value(1)	×	×	×	×	×	✓	✓	✓	✓	✓
Transition	×	×	×	×	×	×	×	×	✓	✓
TransBit	×	×	×	✓	✓	×	✓	✓	✓	✓
TransXor	×	×	✓	×	✓	×	×	✓	✓	✓
TransXorBit	×	✓	✓	✓	✓	×	✓	✓	✓	✓

Figure 4: **Examples of 2-bit expressions and their resistance to the different leakage models.** Each horizontal line corresponds to one bit, and the dashed line separates the two consecutive values for each bit. Variables with name starting with `m` are mask variables, while variables with name starting with `k` are secret variables. A check symbol indicates resistance for the considered leakage model, while a cross indicates a leakage. (1) For the value leakage model, resistance is considered w.r.t. both expressions, i.e. before and after the dashed line.

The rationale behind is that finding a mask verifying the constraint of the algorithm can fail because the expression is not in its most simplified form. On the other hand, calling the `simplify` function takes time which is potentially unnecessary. Therefore, we decided to explore different strategies for calling the `simplify` function, and compare them on an execution time basis as well as on the leakage analysis results. A point which is not mentioned in [4] is if simplifications occur or not between calls to the Algorithm 1, when verifying expressions used as parts of larger expressions. Since this algorithm is destructive regarding the verified expression, the expression must be copied before verification; therefore, we cannot take advantage of a simplification at the start of the verification algorithm in the rest of the user algorithm.

Basically, it is possible to categorize the strategies according to four criteria:

- Simplify or not the expression in the user algorithm (`User`). With SELA z3, it uses the z3 `simplify` function with default parameters, as we tried different values for some of them and observed no change in the results.
- Simplify or not the expression at the beginning of the verification algorithm (`Starting`). For SELA Ex, it is useless if `User` is already used, but it can be useful with SELA z3 in some cases.
- Simplify or not the expression after a replacement, since the expression graph has changed (`Replacement`)
- Simplify or not the expression upon failure and retry verification, if it is the first failure or if the expression has changed since the last retry (`Failure`).

These criteria can be combined in order to make seven different strategies, as summarized

Table 1: **SELA simplification rules.** For all rules, constant `1` refers to the n-bit constant composed of `'1'` only (2^n-1), n being the operator width. *Ext* stands for *ZeroExt* or *SignExt*. In all rules, e, f and g are expressions, while l, m, n, p, q are positive integers.

$e \hat{} 0 \rightarrow e$	$e \hat{} 1 \rightarrow \sim e$	$e \hat{} e \rightarrow 0$	$e \,\&\, 0 \rightarrow 0$	$e \,\&\, 1 \rightarrow e$	$e \,\&\, e \rightarrow e$ | $e \mid 0 \rightarrow e$
$e \mid 1 \rightarrow 1$	$e \mid e \rightarrow e$	$\sim(\sim e) \rightarrow e$	$-(-e) \rightarrow e$	$e - e \rightarrow 0$	$\sim e \hat{} \sim f \hat{} \sim g \rightarrow \sim(e \hat{} f \hat{} g)$
For op $\in$ { &, |, $\hat{}$}: $Ext(n,e)$ op $Ext(n,f) \rightarrow Ext(n, e \text{ op } f)$ $Extract(m,l,e)$ op $Extract(m,l,f) \rightarrow Extract(m,l,e \text{ op } f)$					
$Concat(Extract(m,n,e), Extract(n-1,p,e)) \rightarrow Extract(m,p,e)$ $(m \geq n \geq p)$					
$Extract(p,q,Ext(n,e)) \rightarrow 0$ if $q \geq$ width(e) $\rightarrow Extract(p,q,e))$ if $p \leq$ width(e) $\rightarrow Ext(p$ - width(e), $e)$ if $q = 0$ and $p \geq$width(e) $\rightarrow e$ if $q = 0$ and $p =$ width(e) - 1					
$Extract(p,q,Concat(e_n, ..., e_0))$: remove all e_i which are not included by the indexes p and q; remove *Concat* if p and q are indexes in the same e_i; and remove *Extract* if both p and q coincide respectively with the MSB and the LSB of two internal expressions					

Table 2: **Link between simplification strategies and criteria.** `User` means that the expression is simplified gradually in the user algorithm. `Starting` means that simplify is called at the beginning of the verification. `Replacement` means that simplification is made after each replacement. `Failure` means that when a verification fails, the expression is simplified and the verification is retried. (1) The number of simplifications is limited to one in case of failure.

	Simplification strategy							
	`none`	`u`	`u+s`	`u+r`	`u+f`	`u+s+r`	`u+s+f`	`f1`
`User`		•	•	•	•	•	•	
`Starting`			•			•	•	
`Replacement`				•		•		
`Failure`					•		•	•(1)

in Table 2. We omitted some combinations which we considered not relevant, while the `f1` strategy corresponds to the best of our understanding of the strategy in [4].

3.6 Access to array elements

Accessing an array using as index a symbolic expression is problematic in the general case for performing a verification, since it is not possible to determine which element is accessed. However, in the event of an array implementing a permutation, the access does not change the expression distribution. SELA currently does not support a high-level construct for arrays, but supports an array-type node in graphs: this node type blocks any simplification from occurring between both sides, and otherwise does not change the verification process. In the tested benchmarks, only the AES has array accesses.

Table 3: Experimental Results of SELA analyses

Benchmark	**# Exps Analysed**	**# Leakages Found**		**Execution Time**	
		SELA Ex	SELA z3	SELA Ex	SELA z3
First Order Resistance					
Trichina AND gate v1	10	2	2	0.03s	0.09
Trichina AND gate v2	13	0	0	0.03s	0.09
ISW AND gate 2-sh w/o rand	12	0	0	0.03s	0.08
ISW AND gate 2-sh w/ rand	18	0	0	0.03s	0.10
ISW AND gate 3-sh w/o rand	27	0	0	0.04s	0.11s
ISW AND gate 3-sh w/ rand	45	0	0	0.05s	0.14s
TI AND gate 3-sh	21	0	0	0.04s	0.10s
TI AND gate 4-sh balanced	34	0	0	0.04s	0.10s
GMS AND gate 3-sh	27	0	0	0.04s	0.11s
GMS AND gate 5-sh	72	0	0	0.07s	0.20s
SecMult	359	0	0	6.6s	8.0s
Goubin Conversion	7	0	0	0.07	0.30
AES (KS + 2 rounds)	1050	0	0	–	7m02s
Arm Assembly SecMult	313	0	0	19s	15s
Second Order Resistance					
ISW AND gate 3-sh w/o rand	351	0	0	0.15s	0.83
ISW AND gate 3-sh w/ rand	990	0	0	0.40s	3.0s
GMS AND gate 5-sh	2775	0	0	1.0s	8.3s

4 Experiments

Experiments are composed of three types of benchmarks: circuit masking schemes, masked algorithms and assembly code execution modeled at ISA level. Circuits are composed of the following benchmarks: Trichina AND gate [27] without (v1) and with (v2) additional random; ISW AND gate [16] with two (2-sh) or three (3-sh) shares, and with (w/) or without (w/o) an extra random for computing the terms $a_i b_j$; TI AND gate [21] with three shares (unbalanced) and four shares (balanced); GMS AND gate [22] with three (3-sh) and five (5-sh) shares. The verification is made on each wire, considering a probing security attacker model (*i.e* without glitches). Second order verification is made by considering all the couples of wire expressions.

Masked algorithms ware are composed of the SecMult program [25], Goubin conversion algorithm [13], and the first two rounds of the masked AES [15]. The result of every operation is checked for leakage in the Value model.

Finally, a version of SecMult has been compiled into Arm assembly code, manually translated into SELA data structures, and the effect of each instruction emulated on an ISA model of the processor. For each instruction executed, the leakage was searched for each of the five leakage models presented in Section 3.4 considering the general purpose registers of the processor.

For all experiments, the timeout is set to two hours, noted '–' in tables containing results. All these benchmarks are directly available inside SELA source code archive.

4.1 Leakage Assessment and Performance

Results using the strategy `u+s+r` are shown in Table 3. The column `# Exps Analysed` displays the number of expressions analysed, while the column `# Leakages Found` shows the number of analyses which could not conclude, i.e. the number of potential sources of leakages. These

Table 4: **Breakdown of the experimental results of the assembly version of SecMult.** Number of instructions found leaking for each register in the algorithm. The leakages found were verified to be real leakages w.r.t. their leakage model. Results obtained with the `start_replace` simplify strategy, both for SELA Ex and SELA z3.

Register / Leakage Model	reg 0	reg 1	reg 2	reg 3	reg 4	reg 5	reg 6	reg 7	reg 8	reg 9	reg 12	reg 14
Value	0/26	0/9	0/59	0/93	0/26	0/17	0/4	0/10	0/17	0/9	0/17	0/26
Transition	0/26	0/9	**2**/59	0/93	**1**/26	0/17	0/4	**1**/10	0/17	0/9	0/17	**1**/26
TransBit	0/26	0/9	**1**/59	0/93	**1**/26	0/17	0/4	**1**/10	0/17	0/9	0/17	**1**/26
TransXor	0/26	0/9	**2**/59	0/93	0/26	0/17	0/4	**1**/10	0/17	0/9	0/17	**1**/26
TransXorBit	0/26	0/9	**1**/59	0/93	0/26	0/17	0/4	0/10	0/17	0/9	0/17	**1**/26

columns show that the simplification algorithm and strategy are efficient since the only two potential leakages found are real leakages[2]. Even large code without mask refresh like AES (key schedule and two rounds) and SecMult could be entirely proven leakage-free. Columns `Execution Time` show the execution time on a Intel(R) Xeon(R) E5 2637 v2 @ 3.5GHz for SELA Ex and SELA z3. They show that SELA z3 is more scalable with large expressions (assembly SecMult, AES), but can also slow down the process for small programs, especially when the number of expressions to verify is high. These differences highlight the importance of the internal representation used.

Finally, Table 4 shows the breakdown of the leakages per register and leakage model for the Assembly SecMult. We can observe that the results differ for four of the five models, demonstrating that the choice of the model can have a real impact on the conclusion of the analysis. In the future, we intend to make measurements on a real Arm processor executing this code in order to determine which leakage model is the most relevant in this case, or the sensitivity of each of them, i.e, the number of required traces to exhibit a leakage found by the model.

4.2 Simplification Strategies

We evaluated the different simplification strategies on several representative benchmarks, which do not contain any leakage in the Value leakage model. Results regarding the number of false positives found are presented in Table 5, while execution times are presented in Table 6.

Results from Table 5 show that the simplification strategy has an impact on the number of false positives found. In particular, the most prominent conclusion is that simplifying after a replacement or a failure is mandatory to avoid numerous false positives: strategies `none`, `u` and `u+s` have a lot of false positives for each benchmarks. Then, we can see that only two strategies have no false positive: `u+s+r` and `u+s+f`. Finally, we can notice that simplifying gradually can increase the number of false positives when using SELA z3 (because the z3 simplifications do not always transform the expression in a helpful way regarding verification), while the strategy `f1` can also lead to false positives or timeouts.

Results from Table 6 show that the simplification strategy should include a gradual simplification in order to avoid a performance loss possibly resulting in a timeout. Other than that, the simplification strategy does not influence the execution time much, except for `f1` which can be significantly slower with SELA z3. With the exception of AES on SELA Ex, timeouts only

[2]They have been verified using an enumerative approach

Table 5: Number of expressions incorrectly found to be leaking for the presented simplify strategies, for some of the benchmarks without theoretical leakages in the Value model, and for SELA Ex and SELA z3.

		`none`	`u`	`u+s`	`u+r`	`u+f`	`u+s+r`	`u+s+f`	`f1`
SecMult	# SELA Ex Leakages	102	75	75	0	0	0	0	0
359 Exps	# SELA z3 Leakages	75	89	85	4	4	0	0	0
Arm Asm SecMult	# SELA Ex Leakages	–	83	83	0	0	0	0	–
313 Exps	# SELA z3 Leakages	115	118	114	16	4	0	0	23
GMS AND gate 5-sh	# SELA Ex Leakages	227	227	227	0	0	0	0	0
2775 Exps (2nd order)	# SELA z3 Leakages	227	227	227	0	0	0	0	0
AES (KS + 2 rounds)	# SELA Ex Leakages	–	–	–	–	–	–	–	–
1050 Exps	# SELA z3 Leakages	–	48	48	0	0	0	0	0

Table 6: Execution times for the presented simplify strategies, for some of the benchmarks, and for SELA Ex and SELA z3. The minimum duration of five executions was taken.

		`none`	`u`	`u+s`	`u+r`	`u+f`	`u+s+r`	`u+s+f`	`f1`
SecMult	SELA Ex Time	1h43m	6.4s	6.8s	6.3s	6.2s	6.6s	6.6s	6.2s
359 Exps	SELA z3 Time	20s	6.9s	7.7s	7.6s	7.5s	7.7s	7.7s	18s
Arm Asm SecMult	SELA Ex Time	–	18s	19s	18s	19s	19s	19s	–
313 Exps	SELA z3 Time	33s	14s	15s	15s	16s	15s	15s	52s
GMS AND gate 5-sh	SELA Ex Time	0.9s	0.8s	1.1s	0.8s	0.8s	1.0s	1.0s	0.8s
2775 Exps (2nd order)	SELA z3 Time	9.0s	8.6s	9.3s	7.8s	7.7s	8.3s	8.1s	8.0s
AES (KS + 2 rounds)	SELA Ex Time	–	–	–	–	–	–	–	–
1050 Exps	SELA z3 Time	–	6m53s	7m00	6m43s	6m55s	7m02s	7m04s	9m41s

occur for strategies `none` and `f1` on these benchmarks. Overall, these results show that there is little penalty for doing frequent simplifications, but on the contrary that this results in having few false positives combined with low execution times. Therefore, we would recommend using one of the two strategies `u+s+r` or `u+s+f`.

4.3 Limitations and Directions for Future Work

There is currently no way for enumerating automatically all the k-uplets of a set of n expressions ($k < n$). While this can be done quite easily in python, doing so is not scalable: instead, verifying at higher orders requires more elaborated strategies, as in [3]. Besides, SELA currently does not permit checking for the stronger *Threshold Non-Interference* security property, which consider glitches, as it requires the introduction the notions of *shares*, and not only masks and secrets.

Despite these limitations, which are an axis for future work, SELA provides a strong basic block allowing the verification of symbolic expressions, which can be used for circuits (without glitches), algorithms, or low-level representations like assembly code.

5 Conclusion

We presented SELA, an open python library designed for verifying the absence of leakage in symbolic expressions. We specified the mask selection algorithm which acts as a basis in

the replacement process of the verification. We gave the simplification rules that operate on expressions during the verification, and compared seven different simplification strategies. The results tend to show that if simplifying as much as possible is not always necessary, the cost of simplification remains low, if visible at all, on the tested benchmarks. We introduced five leakage models and illustrated their differences on small examples and on an emulated assembly implementation of SecMult. Finally, we implemented several benchmarks from the literature and showed the accuracy of the implementation, since all leakage-free expressions were proven leakage-free despite the incompleteness of the method, while maintaining low computations times. For these reasons, we believe that the ease of use, performance and accuracy of SELA can make it a good candidate for being used as a base for leakage verification in different contexts.

A first direction for future work consists in adding a support for modelling hardware glitches, and in providing an efficient implementation of higher order verifications. A second direction is to combine the approach of SELA with an inference approach, in order to take advantage of the latter whenever possible, and fall back to the current approach when inference cannot conclude. Finally, a last axis of research concerns the link between leakage models and physical leakages; this includes launching investigations using physical experiments regarding the validity and sensitivity of the leakage models introduced, but more generally finding the level of description required in order to have a correspondence between modeled leakages and real leakages.

References

[1] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014.

[2] 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.

[3] 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.

[4] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. maskverif: a formal tool for analyzing software and hardware masked implementations. *IACR Cryptology ePrint Archive*, 2018:562, 2018.

[5] 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.

[6] Régis Bevan and Erik Knudsen. Ways to enhance differential power analysis. In *International Conference on Information Security and Cryptology*, pages 327–342. Springer, 2002.

[7] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[8] A Adam Ding, Cong Chen, and Thomas Eisenbarth. Simpler, faster, and more robust t-test based leakage detection. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 163–183. Springer, 2016.

[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] 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.

[11] 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.

[12] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[13] 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.

[14] 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.

[15] 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.

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

[17] Yongdae Kim, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. Biasing power traces to improve correlation power analysis attacks. In *First International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2010)*, pages 77–80. Citeseer, 2010.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[19] Thanh-Ha Le, Jessy Clédière, Cécile Canovas, Bruno Robisson, Christine Servière, and Jean-Louis Lacoume. A proposition for correlation power analysis enhancement. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 174–186. Springer, 2006.

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

[21] 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.

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

[23] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 387–399. Springer, 2017.

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

[25] 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.

[26] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 495–513. Springer, 2015.

[27] Elena Trichina. Combinational logic design for aes subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.

[28] 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.