

VerifMSI: Practical Verification of Hardware and Software Masking Schemes Implementations

Quentin L. Meunier¹ ^a and Abdul Rahman Taleb^{1,2}

¹*Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France*

²*CryptoExperts, Paris, France*

Keywords: Side-Channel Attacks, Masking Verification, Threshold Probing Security, Non-Interference.

Abstract: Side-Channel Attacks are powerful attacks which can recover secret information in a cryptographic device by analysing physical quantities such as power consumption. Masking is a common countermeasure to these attacks which can be applied in software and hardware, and consists in splitting the secrets in several parts. Masking schemes and their implementations are often not trivial, and require the use of automated tools to check for their correctness. In this work, we propose a new practical tool named *VerifMSI* which extends an existing verification tool called *LeakageVerif* targeting software schemes. Compared to *LeakageVerif*, *VerifMSI* includes hardware constructs, namely gates and registers, what allows to take glitch propagation into account. Moreover, it includes a new representation of the inputs, making it possible to verify three existing security properties (Non-Interference, Strong Non-Interference, Probe Isolating Non-Interference) as well as a newly defined one called Relaxed Non-Interference, compared to the unique Threshold Probing Security verified in *LeakageVerif*. Finally, optimisations have been integrated in *VerifMSI* in order to speed up the verification. We evaluate *VerifMSI* on a set of 9 benchmarks from the literature, focusing on the hardware descriptions, and show that it performs well both in terms of accuracy and scalability.

1 INTRODUCTION

1.1 Context

Side-Channel Attacks (SCA) exploit the relationship between physical quantities such as power consumption, electromagnetic emissions, or timing information and secret data manipulated by cryptographic implementations, in order to retrieve the secret data. Since the first published differential power attack (Kocher et al., 1999), many other such attacks have proven to be very effective when the device contains no specific countermeasure (Mangard et al., 2008). With the advent of the Internet-of-Things, many embedded devices now use cryptographic implementations and are potential targets for these attacks (smart cards, mobile phones, or RFID tags). Protecting these devices against SCA has thus become a significant concern.

1.2 Background on Masking

Masking is a protection technique against SCA, with a goal to remove the statistical dependency between intermediate computations and secret data manipulated by the program (Trichina, 2003; Ishai et al., 2003). The rationale behind masking is that intermediate computations' values are correlated to the power consumption, therefore a masked program should have no statistical dependency between the secret data and the observable physical quantities. Masking can be applied at any order, and masking at order d consists in splitting each secret variable into $n = d + 1$ parts, called shares. The higher the order, the better the security, as any recombination of up to d shares should not allow to deduce any information on the secret, and the recovery becomes exponentially hard with the number of shares as each observation comes with noise. The most practical and common way of achieving masking for a secret x is the linear masking: it consists in drawing d uniformly and independently distributed variables and computing the n^{th} share by recombining x with the first d shares using the bitwise xor operation \oplus in the boolean case. Splitting

^a  <https://orcid.org/0000-0001-8848-8079>

the secrets is not the main part however, as the original program must then be transformed into a masked equivalent, using shares only, and avoiding any re-combination of the secret variables. While this is quite straightforward for linear operations w.r.t. the xor, the transformation is more complicated for the non-linear parts of the program. When done manually, several security flaws may appear. Consequently, a critical need for automatic masking verification has emerged to check the correctness of masked implementations, both in hardware or software.

Masking is popular as masked implementations can theoretically be proven secure, and several methods have been proposed for proving the security of such implementations. They are all based on analyzing the intermediate expressions manipulated by the circuit or program and try to answer the question: does the distribution of a specific subset of intermediate computation results depend on secret data? The implementation is considered insecure if two different secret values lead to two different distributions. This requires enumerating all possible subsets of internal variables of the program and testing their independence from the secret. The sizes and the number of sets depend on the considered property. One way of testing the independence of a given subset of internal variables is to compute the actual distribution of the variables for each possible realization of the secret inputs. In order to avoid this non-scalable approach, recent works on masking verification use symbolic computation (Ben El Ouahma et al., 2019; Gao et al., 2019; Barthe et al., 2015; Barthe et al., 2019; Meunier et al., 2023). Symbolic verification methods can fail to conclude on some given expressions. In this case, the set of expressions is considered to be “possibly leaking”. Using an enumerative technique to determine the distribution type may help to conclude in this case, but this workaround is limited to small expressions and variable sizes due to the inherent non-scalability of distribution enumeration. Consequently, verification methods must be as accurate as possible to conclude for as many leakage-free expressions as possible and give as few false positives as possible.

1.3 Contributions

LeakageVerif (Meunier et al., 2023) is a verification method implemented in an open-source tool, and provided as a python library. Compared to other tools, it has a good scalability and accuracy, while being easily adapted for different use cases (verification of algorithms, assembly code, hardware modules). In this work, we propose to extend *LeakageVerif* to

overcome some of the limitations of this tool. We thus introduce *VerifMSI* for Verification of Masking Schemes Implementations. We claim *VerifMSI* to be a single tool including state-of-the-art techniques gathering all common masking verification types. If *VerifMSI* does not make a major breakthrough in masking verification techniques, it encompasses a wide range of use cases with optimized and scalable algorithm implementations, making it a very practical open tool for hardware and software masking verification. Compared to *LeakageVerif*, *VerifMSI* makes the following contributions:

- Addition of hardware circuits constructs (gates, registers) allowing for circuits description, taking into account glitches.
- Possibility to use shares for the masking scheme description, allowing to choose between the classical description using secrets and masks, and the share description. In the former, the program explicitly uses secrets and masks, e.g. a secret a is replaced with expressions ma and $ma \oplus a$. In the latter, the shares are atomic inputs of the program (e.g., a_0 , a_1), what allows to verify specific security properties based on shares: Non-Interference (NI), Strong Non-interference (SNI), Probe-Isolating Non-Interference (PINI) and a newly proposed Relaxed Non-Interference (RNI) property which we introduce in this work.
- Higher order verifications of security properties, including optimizations to reduce the number of tuples of expressions to check.

VerifMSI is available as an open-source tool at the following address:

<https://github.com/quentin-meunier/VerifMSI>.

The rest of the article is organised as follows: section 2 presents some background on masking verification and existing security properties; section 3 presents our verification tool *VerifMSI* and the different optimizations we designed; section 4 presents an experimental evaluation of *VerifMSI* on 9 benchmarks from the literature; section 5 compares *VerifMSI* to other existing approaches; finally, section 6 concludes.

2 SECURITY PROPERTIES

2.1 Existing Properties

Since the seminal work of (Ishai et al., 2003) which introduced the first definition of a security property, many other properties were proposed and used. We

recall in the following the most important security notions for hardware circuits.

Threshold Probing Security (TPS). The most common security property targeted with masking is known as *Threshold Probing Security*, for a given order t (Barthe et al., 2015). An implementation achieves t -order threshold probing security if any tuple of intermediate values of size t has a distribution of values which is independent from all secret variables. This security property can reason either on secrets and masks, or on share-based expressions. In the latter case, an expression using shares can be verified by replacing arbitrarily the shares with the corresponding expressions using the secret and masks. The general substitution algorithm is given in algorithm 1. For example, considering two secrets a and b split in two shares (a_0, a_1) and (b_0, b_1) , the expression $a_0 \oplus a_1 \oplus b_0$ is 1-threshold probing secure since b_0 can be replaced with mb or $b \oplus mb$. A replacement in this context is the fact to replace a sub-expression bijective in a mask with the mask itself, which requires the mask to not appear in the sub-expression (step 2 of algorithm 1). In either case, the whole expression of this example is masked with the mask mb , guaranteeing secret independence.

Non-Interference (NI). Another common security property is known as t -order *Non Interference*, or t -NI (Barthe et al., 2019). It is defined informally as the following: an implementation is t -NI if all tuples of t observations (corresponding to internal or output values) have a distribution of values which depends at most on t input shares, for each input. Since we consider the distribution, this allows to make an observation independent from an input share by masking it. Algorithm 1 can also be used for verifying NI with a modified stopping condition, but requires a share description in the implementation. The previous example expression, $a_0 \oplus a_1 \oplus b_0$ is not 1-NI, as it contains two shares of the secret a .

Strong Non-Interference (SNI). Non-Interference can be strengthened to achieve composition, by limiting the number of authorized input shares in each tuple to the number of probes in the tuple which correspond to internal values (as opposed to output values) (Barthe et al., 2019).

Probe Isolating Non-Interference (PINI): is a composable security notion introduced in (Cassiers and Standaert, 2020), which is less restrictive than SNI: a tuple must depend on at most k arbitrary input shares, k being equal to the number of internal probes in the tuple (like SNI), but can also depend on the input shares with the same index as the output shares contained in the tuple.

2.2 Relaxed Non-Interference

The problem with the NI property is that it ignores the masking order when looking at the verification order. Thus, an implementation comprising, among all its expressions, a single one with 2 shares will not even be considered secure at order 1, since for 1-NI, all single expressions should contain at most 1 share occurrence (after masks replacement). This is true even if all inputs are on 3, 4 or more shares, whereas in this case, there cannot be a secret leakage by looking at a single expression. We thus introduce Relaxed Non-Interference (RNI) to solve this problem: informally, it states that for achieving t -order security, all tuples of size t should not contain at least one of the shares for every input (after masks replacement). This definition also allows to remove an implicit condition of Non-Interference which is that all the inputs are split using the same number of shares. As such, we see RNI as an extension of the NI property when the security order is different from the masking order. This is for example the case in Threshold Implementations (Nikova et al., 2006) or the Generalized Masking Scheme (Reparaz et al., 2015).

More formally, we consider an implementation comprising N inputs I_k , each input I_k being split into $d_k + 1$ shares $I_{k_0}, \dots, I_{k_{d_k}}$. Such an implementation is RNI at order t if and only if any tuple of t observations can be perfectly simulated using at most d_k shares for each input I_k (following the notion of perfect simulation in (Belaïd et al., 2016)). For a hardware implementation, it is RNI with glitches if each observation is replaced with the set of input variables it contains in the same combinatorial logic set.

3 VerifMSI

3.1 Overview

VerifMSI is a verification method implementing the substitution algorithm in a python library, seeking to overcome some of the limitations of LeakageVerif. It can thus be seen as an evolution of this tool. Compared to the latter, VerifMSI first adds hardware circuit constructs, allowing to describe circuits with gate and registers, and to take into account glitches in the verification. Second, VerifMSI allows to simply switch between a share-based and a secrets and masks based description, allowing to verify the NI, SNI, PINI and the proposed RNI properties as well as TPS. Third, VerifMSI implements optimizations in order to reduce the number of probes in the circuit,

Algorithm 1: Substitution algorithm for verifying threshold probing security, from (Barthe et al., 2018).

procedure THRESHOLDPROBINGSECURITY(e)

Inputs: tuple of expressions $V = (v_1, \dots, v_n)$, 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.

allowing it to efficiently perform higher order verifications.

Figure 1 shows a code fragment of a VerifMSI program for implementing a first order Domain Oriented Masking (DOM) AND circuit (Groß et al., 2017), and the associated circuit in Figure 2. Secrets are declared on lines 1 and 2, while their sharing is done on lines 6 and 7. The `getRealShares` function returns a specified number of shares of a secret, which are not equivalent to a secret and mask representation. Alternatively, one can use the `getPseudoShares` function, which does a sharing using secret and masks (typically $(m_0, k \oplus m_0)$ at order 1). The latter representation is useful for verifying TPS. Line 3 declares a 1-bit mask, while lines 10 to 14 create input gates associated to the inputs. Lines 17-20 make all the cross products between shares; note the gates are n -ary and can take an arbitrary number of parameters. Lines 23 to 30 implement the remaining gates and registers: registers stop the propagation of glitches. Indeed, without registers, a gate can leak all of its input wires (cf. Figure 2). Finally, line 33 checks the NI property on the outputs c_0 and c_1 , at order `order` (here `order` should be 1), with or without glitches according to the `withGlitches` parameters.

3.2 Optimisations

In order to reduce the number of tuples verified, especially for higher orders, VerifMSI implements some optimisations for hardware descriptions consisting in removing some of the observations, which for the most part are based on the optimisations made in (Belaïd et al., 2022). These optimisations are based on the fact that we do not just verify expressions, but a circuit, or gadget, allowing us to make additional assumptions. For instance, it is always possible to observe single input shares, what allows us to remove them when enumerating the tuples, by considering partial tuples (Belaïd et al., 2022). The optimisations implemented are the following:

- Removal of observations constituted of at most one share per input and no random (optimisation **v0**).

```

1 a = symbol('a', 'S', 1) # 1-bit
  secret
2 b = symbol('b', 'S', 1) # 1-bit
  secret
3 z10 = symbol('z10', 'M', 1) # 1-bit
  mask
4
5 # Do the sharing for 'a' and 'b'
6 a0, a1 = getRealShares(a, 2)
7 b0, b1 = getRealShares(b, 2)
8
9 # Create input gates
10 a0 = inputGate(a0)
11 a1 = inputGate(a1)
12 b0 = inputGate(b0)
13 b1 = inputGate(b1)
14 z10 = inputGate(z10)
15
16 # Cross products
17 a0b0 = andGate(a0, b0)
18 a0b1 = andGate(a0, b1)
19 a1b0 = andGate(a1, b0)
20 a1b1 = andGate(a1, b1)
21
22 # Remaining gates and registers
23 a1b0 = xorGate(a1b0, z10)
24 a1b0 = Register(a1b0)
25 a0b1 = xorGate(a0b1, z10)
26 a0b1 = Register(a0b1)
27 c0 = a0b0
28 c0 = xorGate(c0, a0b1)
29 c1 = a1b1
30 c1 = xorGate(c1, a1b0)
31
32 # Check the NI security property
33 checkSecurity(order, withGlitches,
  'ni', c0, c1)

```

Figure 1: Example of VerifMSI program implementing a first order DOM AND circuit.

- Removal of observations which are redundant with some others (optimisation **v1**). An expression e_0 is considered redundant and is omitted when there exists an expression e_1 such that:
 - all the mask occurrences in e_0 (resp. e_1) are bijective occurrences w.r.t. e_0 (resp. e_1);
 - e_0 and e_1 have the same mask occurrences;

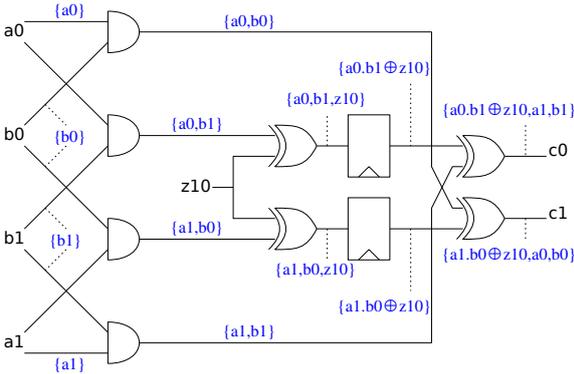


Figure 2: Order 1 DOM AND circuit from (Groß et al., 2017), with the leakage associated to each wire when glitches are considered.

Table 1: Breakdown of observations removal in VerifMSI for the ISW AND implementation, for N-share inputs. The first column gives the expression forms which can be removed for one of the presented simplifications, the second column gives the number of occurrences, and the following columns which simplification rule allows to remove the corresponding expressions.

Expression form	# Occ.	v0	v1
a_i, b_i	$2N$	✓	
$a_i.b_j$	N^2	✓	
$z_{i,j}$	$N(N - 1) / 2$		✓
$a_i.b_j \oplus z_{i,j}$	$N(N - 1) / 2$		✓
$a_0.b_j \oplus a_j.b_0 \oplus z_{0,j}$	$N - 1$		✓
$a_0.b_0 \oplus z_{0,1}$	1		✓
# Removable Obs.	$2N(N + 1)$		
# Total Obs.	$2N + N^2 + 5N(N - 1)/2$		
# Remaining Obs.	$N(3N - 5)/2$		

- All the input shares appearing in $e0$ also appear in $e1$.

Table 1 details the effects of these optimisations in terms of number of removed observations, taking as example the ISW AND.

We can notice that the number of removable observations thanks to both optimisations is significant, and even crucial, as for example with a 7-share inputs ISW AND circuit, it allows to go from 168 intermediate values to 56. Asymptotically, the reduction still allows to remove 57% of the probes.

Finally, we can notice that when glitches are considered, the number of probes to keep for enumeration is largely reduced, as the probes corresponding to wires which are not preceding a register can be ignored: the leakage associated with them will be part of the leakage associated with the output of the next gate.

3.3 Improving the Mask Choice for Replacements

While running through the process of benchmarking, we encountered a few false positives in the verification of one benchmark (ISW AND), i.e. a potential leakage was reported by VerifMSI while there was actually none. After investigation, it appeared that in these cases, the sequence of mask selections for replacement led to the impossibility to conclude, whereas another sequence of choices would allow it.

Going into the details, we noticed two distinct problems. First, the algorithm in Figure 1 allows a mask to be taken only once, in order to guarantee termination. Yet, we encountered some cases in which taking an already taken mask is necessary in order to conclude. This can happen when the mask originally has several occurrences, and after some replacements and simplifications, only has a single occurrence. In order to take this into account while still guaranteeing termination, we authorize a mask to be taken several times only if it has a single occurrence. Since the expression necessarily decreases in size during a replacement using a mask having a single occurrence, this can happen only a finite number of times.

The second false positive problem we noticed happened when selecting for a replacement a mask being itself an element of the tuple. However, when removing entirely the possibility to select such masks for replacements, other failures were reported. Therefore, we modified the mask selection algorithm to make it possible to select such masks for replacement, but with the lowest priority. Using this heuristic, no false positives due to mask selection arose in any benchmark.

4 EXPERIMENTAL EVALUATION

We perform an evaluation of VerifMSI on several benchmarks from the literature. We focus the evaluation on hardware circuits as the software implementations descriptions are similar to those of LeakageVerif. The experiments comprise the following programs:

- ISW AND: The logical AND masking scheme (Ishai et al., 2003)
- ISW AND refresh: A combination of the ISW AND with a circular refresh on one of the input (De Cnudde et al., 2016)
- DOM AND: The Domain Oriented Masking implementation of the AND gate (Groß et al., 2017), resistant to glitches;

- Refresh $N \log N$: The $N \log N$ refresh scheme (Battistello et al., 2016);
- NI Mult and SNI Mult: The NI and SNI multiplication schemes (Bordes and Karpman, 2021);
- PINI Mult: The PINI multiplication scheme (Wang et al., 2023);
- GMS AND: Two implementations of the AND gate using the Generalized Masking Scheme, described in the article, using respectively 3 and 5 shares (Reparaz et al., 2015);
- TI AND: The balanced Threshold Implementation of the AND gate (Nikova et al., 2006).

All benchmarks were run on a single core on a server with an Intel CPU Xeon E5-2637v2@3.5GHz, under the CentOS 9 operating system. For all benchmarks, we set a timeout to 6 hours, and a memory limit to 110 GB (which was never reached).

Table 2 presents the verification results of the different circuits we implemented. All of these circuits implementations are provided along with the code of VerifMSI. Only the configurations for which the verified property was known to be true were run, and only the configurations which did not exceed the 6 hours timeout are presented in the table. The verification order is set to the designed security order: this is always the number of shares minus one, except for GMS AND with 3 shares (order 1), GMS AND with 5 shares (order 2) and TI AND (order 1).

From the results in table 2, we can make the following observations: VerifMSI can verify all the pre-selected hardware masking schemes, up to a certain order (between 5 and 7 shares). We notice that the verification of the TPS property scales significantly less than the other properties, due to the fact that the optimisation targeting the reduction in the number of probes do not apply with a representation using secrets and masks.

We also notice that the GMS AND and TI AND circuits can only be verified with TPS and RNI, as the order of security is not equal to the number of shares minus one – they typically contain tuples of size 1 depending on several input shares. This underlines the interest of the RNI property, which is the only property based on shares adapted to the verification of such masking schemes.

Finally, we can see that there are a few false positives on PINI mult with five shares or more. Analysing them in more details reveals that the tuple does not contain anymore mask, and that the problem occurs because VerifMSI is not able to factorize an expression and make some products disappear.

5 RELATED WORKS

A certain number of tools target the verification of security properties in masked software or hardware implementations.

LeakageVerif. LeakageVerif (Meunier et al., 2023) is a flexible and open-source verification tool achieving good accuracy and scalability, provided as a python library. LeakageVerif can verify implementations at different abstraction levels (algorithmic, code, assembly, circuit), but can only verify threshold probing security on a description using secrets and masks. Moreover, it cannot take glitches into account in hardware descriptions. The fact that the tool is provided as a python library allows to have simulable descriptions, and to support all python’s control mechanisms. **MaskVerif.** maskVerif (Barthe et al., 2019) is a tool written in OCaml designed for the verification of circuits. The strength of maskVerif is its ability to scale well with higher orders. However, it is not very well adapted for software masking schemes implementations, as it lacks support for arithmetic operations, arbitrary size variables and bit concatenation and extraction operations. Finally, maskVerif does not permit to express a non-linear control flow, allowing only for function calls.

IronMask. IronMask (Belaïd et al., 2022) is an open-source tool designed for the verification of masked hardware implementations. The tool has an excellent scalability due to its optimized writing in C, and can verify many security properties. On the downside, it is not able to verify TPS, and is limited to certain types of implementations in which the masks must be linearly added to given shares.

SILVER. SILVER (Knichel et al., 2020) is a tool able to verify common security properties on hardware descriptions. It takes as input either a Verilog implementation or an instruction list and checks the TPS, NI, SNI and PINI notions with or without glitches, as well as the uniformity of some output sharing. The tool suffers however from a limited scalability.

If VerifMSI is not the fastest of these tools for most hardware implementations and configurations at high orders, it is the only tool which can verify all common security properties, using both share-based and secrets and masks descriptions, for both hardware and software masking schemes, and having the benefits of using all of the python constructs.

Table 2: Verification times and number of tuples verified with VerifMSI, for higher order hardware masking schemes from the literature. Column #Sh. indicates the number of shares, and Time the verification time. w/ g. means “with glitches”, and values between parenthesis indicate the number of tuples for which the verification failed (false positives).

Gadget	#Sh.	Property	Time	# Tuples	Gadget	#Sh.	Property	Time	# Tuples		
Refresh N log N	5	TPS	9s	23751	ISW AND	4	TPS	12s	24804		
		NI	2s	7546			NI	<1s	469		
		SNI	2s	7546			SNI	<1s	469		
		RNI	2s	7546			RNI	<1s	469		
		PINI	2s	7546			PINI	<1s	469		
	6	TPS	7m23s	850668		TPS	25m15s	2024785			
		NI	1m58s	284273		NI	8s	15275			
		SNI	1m58s	284273		SNI	8s	15275			
		RNI	1m58s	284273		RNI	8s	15275			
7	PINI	118s	284273	PINI	8s	15275					
	TPS	3h50m31	20358520	6	NI	8m31s	667927				
	NI	47m16s	5358577		SNI	9m4s	667927				
	SNI	48m30s	5358577		RNI	8m41s	667927				
	RNI	47m17s	5358577		PINI	8m36s	667927				
PINI	47m53s	5358577									
ISW AND refresh	3	TPS	<1s	741	PINI Multi- plication	4	TPS	13s	24804		
		NI	<1s	325			NI	<1s	469		
		SNI	<1s	325			RNI	<1s	469		
		RNI	<1s	325			PINI	<1s	469		
		PINI	<1s	325			5	TPS	26m36s	2024785 (98)	
	4	TPS	27s	45760	NI	7s		15275 (1)			
		NI	5s	17343	RNI	7s		15275 (1)			
		SNI	6s	17343	PINI	7s		15275 (1)			
		RNI	5s	17343	6	NI		7m24s	667927 (77)		
	PINI	5s	17343	RNI		7m20s	667927 (79)				
	5	TPS	1h1m	3921225		PINI	7m29s	667927 (79)			
		NI	10m54s	1356201		NI Multi- plication	5	TPS	10m21s	916895	
		SNI	12m3s	1356201	NI			<1s	385		
		RNI	10m49s	1356201	RNI			<1s	385		
		PINI	10m46s	1356201	PINI			<1s	385		
DOM AND	5	TPS	45m36s	4780230	6			7	NI	36	55454
		NI	25s	59535		RNI	36s		55454		
		SNI	26s	59535		PINI	37s		55454		
		RNI	25s	59535		5	5		NI	28m19s	2007327
		PINI	26s	59535					RNI	28m34s	2007327
		TPS w/ g.	15s	12650	PINI			28m52s	2007327		
		NI w/ g.	8s	12650	SNI Multi- plication			6	TPS	26m18s	2024785
		RNI w/ g.	9s	12650					NI	7s	15275
		PINI w/ g.	9s	12650		SNI	8s		15275		
		6	NI	35m19s		3505050	RNI		7s	15275	
	SNI		36m18s	3505050		PINI	8s		15275		
	RNI		35m42s	3505050	3	GMS AND	TPS	<1s	30		
	PINI		38m1s	3505050			RNI	<1s	9		
	TPS w/ g.		11m35s	376992			5	TPS	1s	3570	
	NI w/ g.		6m11s	376992				RNI	<1s	630	
RNI w/ g.	6m11s		376992	4	TI AND	TPS	<1s	34			
PINI w/ g.	6m23s	376992	RNI			<1s	4				
7	NI w/ g.	5h13m	13983816								
	RNI w/ g.	5h11m	13983816								
	PINI w/ g.	5h15m	13983816								

6 CONCLUSION AND FUTURE WORK

We presented VerifMSI, a practical tool implemented as a python library for verifying masking schemes implementations. It extends the existing LeakageVerif tool with constructs targeting hardware implementations, and enriches it with the verification of four security properties (NI, SNI, RNI, PINI). The experiments presented in the article, focusing on 9 hardware schemes, show that VerifMSI is able to successfully verify many implementations from the literature, for masking orders of up to 7 shares.

Future work includes enriching the software side of VerifMSI with support for Galois Field operations, as well as implementing less common security properties, and in particular the ones defined in the random probing model. We also plan to write the core of VerifMSI in a compiled language to reduce the cost of enumeration.

REFERENCES

- Barthe, G., Belaïd, S., Cassiers, G., Fouque, P.-A., et al. (2019). maskverif: Automated verification of higher-order masking in presence of physical defaults. In *European Symposium on Research in Computer Security*, pages 300–318. Springer.
- Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., et al. (2015). Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer.
- Barthe, G., Belaïd, S., Fouque, P.-A., and Grégoire, B. (2018). maskverif: automated analysis of software and hardware higher-order masked implementations. Technical report, 562.
- Battistello, A., Coron, J.-S., Prouff, E., and Zeitoun, R. (2016). Horizontal side-channel attacks and countermeasures on the isw masking scheme. In *Cryptographic Hardware and Embedded System*, pages 23–39. Springer.
- Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., et al. (2016). Randomness complexity of private circuits for multiplication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 616–648. Springer.
- Belaïd, S., Mercadier, D., Rivain, M., and Taleb, A. R. (2022). Ironmask: Versatile verification of masking security. In *IEEE Symposium on Security and Privacy*, pages 142–160. IEEE.
- Ben El Ouahma, I., Meunier, Q. L., Heydemann, K., and Encrenaz, E. (2019). Side-channel robustness analysis of masked assembly codes using a symbolic approach. *Journal of Cryptographic Engineering*, pages 231–242.
- Bordes, N. and Karpman, P. (2021). Fast verification of masking schemes in characteristic two. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 283–312. Springer.
- Cassiers, G. and Standaert, F.-X. (2020). Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, pages 2542–2555.
- De Cnudde, T., Reparaz, O., Bilgin, B., Nikova, S., et al. (2016). Masking aes with shares in hardware. In *Cryptographic Hardware and Embedded Systems*, pages 194–212. Springer.
- Gao, P., Zhang, J., Song, F., and Wang, C. (2019). Verifying and quantifying side-channel resistance of masked software implementations. *ACM Transactions on Software Engineering and Methodology*, pages 1–32.
- Groß, H., Mangard, S., and Korak, T. (2017). An efficient side-channel protected aes implementation with arbitrary protection order. In *Topics in Cryptology—CT-RSA 2017: The Cryptographers’ Track at the RSA Conference*, pages 95–112. Springer.
- Ishai, Y., Sahai, A., and Wagner, D. (2003). Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer.
- Knichel, D., Sasdrich, P., and Moradi, A. (2020). Silver–statistical independence and leakage verification. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 787–816. Springer.
- Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer.
- Mangard, S., Oswald, E., and Popp, T. (2008). *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media.
- Meunier, Q. L., Pons, E., and Heydemann, K. (2023). Leakageverif: Scalable and efficient leakage verification in symbolic expressions. *IEEE Transactions on Software Engineering*.
- Nikova, S., Rechberger, C., and Rijmen, V. (2006). Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer.
- Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., and Verbauwhede, I. (2015). Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer.
- Trichina, E. (2003). Combinational logic design for aes subbyte transformation on masked data. *Cryptology EPrint Archive*.
- Wang, W., Ji, F., Zhang, J., and Yu, Y. (2023). Efficient private circuits with precomputation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 286–309.