SYSTEMATIC
PARIS REGION SYSTEMS & ICT CLUSTER

VERISICC
Deliverable 2.1:
Evaluation of existing solutions

APPEL A PROJET: FUI25

ACRONYME DU PROJET: VERISICC

NOM DU PROJET: VERISICC

Leader of this deliverable

- Partner: University of Luxembourg
- Nom du contact: Jean-Sébastien Coron
- Contact information: jean-sebastien.coron@uni.lu

Leader of the project

- Company: CryptoExperts
- Contact name: Sonia Belaïd
- Contact information: sonia.belaid@cryptoexperts.com, 06 68 75 30 66

Partners

- SMEs: CryptoExperts and NinjaLab
- Big Business: IDEMIA
- Public Institutions: INRIA, ANSSI, and Université du Luxembourg

Version : June 2020

Sommaire

1 Introduction **3**
- 1.1 The masking countermeasure . . . 3
- 1.2 Security proofs . . . 3

2 Formal verification tools of masking **4**
- 2.1 Verification of first-order masking schemes . . . 4
- 2.2 Verification of higher-order masking schemes . . . 5
  - 2.2.1 `maskVerif` . . . 5
  - 2.2.2 `CheckMasks` . . . 5

3 Needs from the industry **6**

4 The `CheckMasks` approach **6**
- 4.1 Program representation and formal manipulations . . . 7
- 4.2 Identification of side-channel vulnerabilities . . . 8
- 4.3 Taking into account arithmetic operations . . . 8

5 The `maskVerif` approach **9**
- 5.1 Program representation . . . 10
- 5.2 Identification of side-channel vulnerabilities . . . 12
- 5.3 Taking into account arithmetic operations . . . 12

1. Introduction

1.1. The masking countermeasure

Masking is the most widely used countermeasure against side-channel attacks for block-ciphers and symmetric-key algorithms. In a first-order countermeasure, all intermediate variables x are masked into a pair (x', r) where r is a randomly generated value and $x' = x \oplus r$. For such countermeasure, it is usually straightforward to verify its security against first-order attacks; namely it suffices to check that all intermediate variables have the uniform distribution, or at least that their distribution is independent from the key; therefore an attacker processing the side-channel leakage of intermediate variables separately (as in a first-order attack) does not get useful information.

However second-order attacks combining the leakage on x' and r can be mounted in practice, so it makes sense to design masking algorithms resisting higher-order attacks. This is done by extending Boolean masking to n shares with $x = x_1 \oplus \cdots \oplus x_n$; in that case an implementation should be resistant against t-th order attacks, in which the adversary combines leakage information from at most $t < n$ intermediate variables.

1.2. Security proofs

In principle any countermeasure against high-order attacks should have a security proof, but such proof can be either missing, incomplete, or incorrect.

The first step is to specify what it means for a masking countermeasure to be secure, *i.e.* what is the security model. Such formalization was initiated by Ishai, Sahai and Wagner in [ISW03]. In this model, the adversary can probe at most t wires in the circuit, but he should not learn anything about the secret key. The approach for proving security is based on simulation: one must show that any set of t wires probed by the adversary can be perfectly simulated without the knowledge of the secret-key. This shows that the t probes do not bring any useful information to the attacker, since he could run this simulation by himself.

More precisely, the simulation technique consists in showing that any set of t probes can be perfectly simulated by the knowledge of only a proper subset of the input shares x_i. At the beginning of the algorithm an original variable x is shared into n shares x_i. When x is part of the secret-key, this pre-sharing cannot be probed by the adversary. Since any subset of at most $n-1$ input shares x_i are uniformly and independently distributed, the simulation of the probed variables can be performed without knowing the secret-key.

The main result in [ISW03] is to show that any circuit C can be transformed into a new circuit C' of size $\mathcal{O}(t^2 \cdot |C|)$ that is resistant against an adversary probing at most t wires in the circuit. The construction is based on secret-sharing every variable x into n shares with $x = x_1 \oplus \cdots \oplus x_n$, and processing the shares in a way that prevents a t-limited adversary from learning any information about the initial variable x, using $n \geq 2t + 1$ shares.

One advantage of the ISW model is that its conceptual simplicity makes it amenable to formal verification. This has been demonstrated in a series of works, including [MOPT12, BRNI13, EWS14, BBD+15a, BBD+16a, **?**, ZGSW18, BIKM18].

The most immediate benefit of formal verification is its automation, allowing to deal with the combinatorial complexity of proving masked implementations secure. This complexity is specially significant for implementations where secrets are split into a large number of shares; we call such

implementations higher-order. Perhaps more importantly, formal verification has also been instrumental for advancing the state-of-the-art in masking. First, formal verification tools have been used to reduce the randomness cost of existing schemes. Second, strong non-interference, which solves a long-standing problem of compositional reasoning for masking, has first emerged in the context of formal verification, before being adopted in the literature on masking.

Another model is the noisy leakage model introduced by Chari et al. [CJRR99] then extended by Prouff and Rivain [PR13]. It describes many realistic side-channel attacks where an adversary obtains each intermediate value perturbed with a "δ-noisy" leakage function. A leakage function L is called δ-noisy if for a uniformly random variable Y we have $\mathrm{SD}(Y; Y|\boldsymbol{L}_Y) \leq \delta$, with SD the statistical distance. It was shown in [DFS15] that an equivalent condition is that the leakage is not too informative, where informativity is measured with the standard notion of mutual information $\mathrm{MI}(Y; \boldsymbol{L}_Y)$. In contrast with the ϵ-probing model, the adversary obtains noisy leakage for each intermediate variable. For example, in the context of masking, he obtains L$(Y_i, \boldsymbol{R}_i)$ for all the shares Y_i, which is reflective of actual implementations where the adversary can potentially observe the leakage of all these shares, since they are all present in leakage traces.

A third and intermediate model is getting more and more important in the literature recently, the random probing model. Basically, it assumes that every wire or intermediate variable in a circuit leaks with some probability p. The circuit is then secure if the probability to leak intermediate variables which jointly depend on the secrets is negligible. This model benefits from being more realistic than the probing model since it captures a larger set of attacks. For instance, horizontal attacks that may exploit the multiple use of a variable to get information on its value is not captured in the probing model while it is handled in the random probing model where the probability to get the value is increased by the repetition.

Duc et al. showed that security against probing attacks implies random probing security for some probability which itself implies security against noisy leakages [DDF14]. This result leads to the natural strategy of proving security in the (simpler) probing model while stating security levels based on the concrete information leakage evaluations (as discussed in [DFS15]).

2. Formal verification tools of masking

A first sequence of work manages to provide verification of masked implementations, but is mainly restricted to first-order masking schemes. We give an overview below and then focus on recent tools that achieve verification of higher-order verification schemes.

2.1. Verification of first-order masking schemes

In 2012, Moss et al. [MOPT12] implement the first automated method to verify and build masked implementations. Basically, they design a type-based masking compiler that tracks variables that are masked by random values and iteratively modifies an unprotected program until all secrets are masked. This strategy achieves first-order probing security. Although efficient and scalable, this method is overly conservative and rejects secure programs.

Bayrak et al. [BRNI13] investigate in 2013 the logic-based verification which offers interesting trade-offs between efficiency and expressiveness. Their SMT-based method is able to analyze the probing security of first-order masked implementations by proving statistical independence between secrets and leakage. It was later extended by Eldib, Wang and Schaumont [EWS14] to achieve

higher-order masking verification with a method based on model counting. The authors use incremental verification in order to circumvent the resulting exponential blow-up. Despite these improvements, the scope of the method remains limited on concrete examples. Finally, Zhang et al. [ZGSW18] provide additional improvements in terms of precision and scalability with their tool SCInfer (partly inspired from `maskVerif` below).

2.2. Verification of higher-order masking schemes

In the following, we focus only on recent tools able to provide verifications of concrete masking schemes at higher orders.

2.2.1. `maskVerif`

In 2015, Barthe et al. exhibited an automated method to prove the security of masked implementation against t-th order attacks, for small values of t [BBD+15a]. The method only works for small values of t because the number of possible t-tuples of intermediate variables grows exponentially with t. To formally prove the security of a masking algorithm, the authors describe an algorithm to construct a bijection between the observations of the adversary (corresponding to a t-tuple of intermediate variables) and a distribution that is syntactically independent from the secret inputs; this implies that the adversary learns nothing from this particular t-tuple of intermediate variables. All possible t-tuples of intermediates variables are then examined by exhaustive search.

The authors obtain a formal verification of various masked implementations, up to second order masked implementation of AES, and up to 5-th order for the masked Rivain-Prouff multiplication [RP10a]. In particular, the authors were able to rediscover some known attacks and discover new ways of attacking already broken schemes.

The main drawback of this approach is that it can only work for small orders t and small programs, since the running time is exponential in t and the size of the program.

The tool allows to check various security properties, like probing security, non-interference (NI) and strong non-interference (SNI). It is also able to perform security analysis in presence or in absence of glitches from its extension in 2019, or with transition or without transition. It provides a small intermediate language for describing implementation but it can also take Verilog implementation as input.

2.2.2. `CheckMasks`

A simplification and extension of the formal verification results from [BBD+15a] and [BBD+16a] was described in [**?**]. Two complementary approaches were described: a generic approach for the formal verification of any circuit, but for small attack orders only (as in [BBD+15a]), and a specialized approach for the verification of specific circuits, but at any order (as in [BBD+16a]).

For the generic verification of countermeasures at small orders, the author uses a different formal language from [BBD+15a]. In particular the underlying circuit is represented as nested lists, which leads to a simple and concise implementation in Common Lisp, a programming language well suited to formal manipulations. The author is then able to formally verify the security of the Rivain-Prouff countermeasure with very few lines of code. The running times for formal verification are similar to those in [BBD+15a]. Thanks to this simpler approach, the author also extends [BBD+15a] to handle a combination of arithmetic and Boolean operations, and formally verifies the security of the recent

Boolean to arithmetic conversion algorithm from [Cor17b]. This approach is implemented in a tool, referred to as `CheckMasks`.

For the verification of specific gadgets at any order (instead of small orders only with the generic approach), the technique is quite different from [BBD+16a] and consists in applying elementary transforms to the circuit, until the t-NI or t-SNI properties become straightforward to verify. For a set of well-chosen elementary transforms, the formal verification time becomes polynomial in t (instead of exponential with the generic approach); this implies that the formal verification can be performed at any order. The `CheckMasks` tool provides a formally verified proof of the t-SNI property of the multiplication algorithm in the Rivain-Prouff countermeasure, and of the mask refreshing based on the same multiplication algorithm; in both cases the running time of the formal verification is polynomial in the number of shares n.

Finally, the author shows how to get the best of both worlds, at least for simple circuits: he shows how to automatically apply the circuit transforms that lead to a polynomial time verification, based on a limited set of generic rules. Namely he identifies a set of three simple rules that enable to automatically prove the t-SNI property of the multiplication based mask refreshing, and also two security properties of mask refreshing considered in [Cor17b]. The source code of `CheckMasks` verification tool is publicly available at [Cor17a], under the GPL v2.0 license.

3. Needs from the industry

Strong performance constraints exist for embedded implementations. Namely embedded processors usually have a relatively low frequency compared to PC processors, a low amount of memory, and a slow TRNG. In general the goal is to minimize the cost of the processor, and its energy consumption. For best performances, cryptographic algorithms are most often implemented in C or even in assembly for critical parts. From the deliverable “L1.2: Définitions des besoins”, the following needs have been identified by the industrial partners:

1. Work directly from assembly language implementation.
2. Better identify the side-channel vulnerabilities: when a leakage occurs, the exact position of the leakage should be determined.
3. Take into account all low-level operations used in an implementation (not only Xors), such as for example arithmetic operations in Boolean to arithmetic conversions, and look-up tables.

4. The `CheckMasks` approach

The generic verification of the masking countermeasure was initiated by Barthe *et al.* in [BBD+15b] based on the EasyCrypt framework. The approach consists in considering all possible t-tuples of intermediate variables, and proving that for each particular t-tuple the adversary learns nothing from the secret key. The authors obtained a formal verification of various masked implementations, up to second order masked implementation of AES, and up to 5-th order for the masked Rivain-Prouff multiplication [RP10b].

In this section we consider the CheckMasks tool described in [**?**]; the source code of the CheckMasks library is publicly available at [Cor17a], under the GPL v2.0 license. The circuit to be verified is represented as nested lists, which leads to a simple and concise implementation in Common Lisp, a

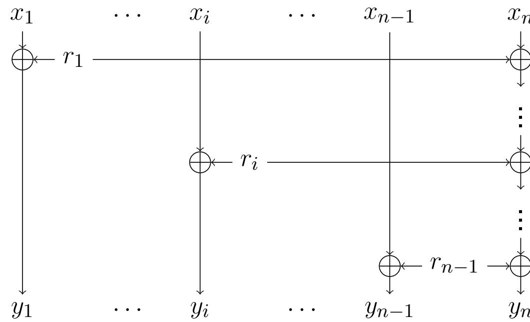

Figure 1: The RefreshMasks algorithm, with the randoms r_i accumulated on the last column.

programming language well suited to formal manipulations. In [**?**] a formal verification of the security of the Rivain-Prouff multiplication is described, with running times similar to those in [BBD+15b].

4.1. Program representation and formal manipulations

In CheckMasks, a circuit is represented with nested lists, using the prefix notation. Consider for example the circuit taking as input x and y and outputting $x \oplus y$; we represent it as (+ X Y). Similarly the circuit computing $x \cdot y$ is represented as (* X Y). To represent more complex circuits the lists are recursively nested. For example, to represent the circuit $x \cdot (y \oplus z)$, we write (* X (+ Y Z)). If a circuit has many outputs, we represent the list of outputs without any prefix operator; for example, the circuit outputting $(x \oplus y, x \cdot y)$ can be represented as ((+ X Y) (* X Y)).

Consider for example the RefreshMasks algorithm, as illustrated in Fig. 1. It is easy to write a program in Common Lisp that generates the circuit corresponding to RefreshMasks; we refer to [Cor17a] for the source code. For example, we obtain for $n = 3$ input shares:

```
> (RefreshMasks '(X1 X2 X3))
((+ R1 X1) (+ R2 X2) (+ R2 (+ R1 X3)))
```

which corresponds to $y_1 = r_1 \oplus x_1$, $y_2 = r_2 \oplus x_2$ and $y_3 = r_2 \oplus (r_1 \oplus x_3)$. Note that the above RefreshMasks function in Common Lisp takes as input a list of n shares (here $n = 3$) and outputs a list of n shares; therefore it can be easily composed with other such Common Lisp functions to create more complex circuits.

In order to verify a security property of a gadget, one performs a sequence of elementary substitutions. Consider for example the two output variables (+ R1 X1) and (+ R2 (+ R1 X3)) from above. We would like to show that these two variables are uniformly and independently distributed. Since the random R2 is used only once in those two outputs, it can play the role of a one-time pad, and we can perform the following substitution in the second output:

$$\text{(+ R2 (+ R1 X3))} \longrightarrow \text{R2}$$

Namely, since R2 is used only once, the distribution of (+ R2 (+ R1 X3)) is the same as the distribution of R2. More generally, when a random R is used only once, we can perform the substitution:

$$\text{(+ R X)} \longrightarrow \text{R} \tag{1}$$

Starting with the above list of two output variables, we can perform the following sequence of elementary substitutions:

$$\text{((+ R1 X1) (+ R2 (+ R1 X3)))} \longrightarrow \text{((+ R1 X1) R2)} \longrightarrow \text{(R1 R2)}$$

The first substitution is possible because R2 is used only once, and the second substitution is possible because R1 is used only once after the first substitution. Since we have obtained two distinct randoms (R1 R2) at the end, the two output variables are uniformly and independently distributed, as required.

In CheckMasks, the security properties are verified by performing a sequence of elementary transformations of a circuit. These transformations are generally easy to implement in Common Lisp. However, it seems difficult for the CheckMasks tool to work directly with an assembly language implementation of a cryptographic algorithm, as in such representation a formal transformation such as (1) would be much harder to perform.

4.2. Identification of side-channel vulnerabilities

In the CheckMasks tool, side-channel vulnerabilities are generally easy to identify. Consider for example the above RefreshMasks algorithm. While the RefreshMasks algorithm achieves the t-NI property, it is easy to see that it does not achieve the stronger t-SNI property, as already observed in [BBD+16b]. Namely one can probe the output $y_1 = r_1 \oplus x_1$ and the internal variable $y_{n,1} = r_1 \oplus x_n$; see Fig. 1. This gives $y_1 \oplus y_{n,1} = x_1 \oplus x_n$ and therefore the knowledge of both inputs x_1 and x_n is required for the simulation, while only $t = 1$ internal variables has been probed (and therefore, at most one input x_i can be used for the simulation to achieve the t-SNI property). More generally, using the CheckMasks formal tool, we can obtain the list of all $(n-1)$-tuples of probes that break the t-SNI property of RefreshMasks with n input shares. For example, we obtain for $n = 4$:

```
> (Check-Refreshmasks-Non-SNI 4 :all 't)
Input: (X1 X2 X3 X4)
Output: ((+ R1 X1) (+ R2 X2) (+ R3 X3) (+ R3 (+ R2 (+ R1 X4))))
((+ R1 X1) X2 (+ R1 X4))
((+ R1 X1) (+ R2 X2) (+ R1 X4))
((+ R1 X1) (+ R2 X2) (+ R2 (+ R1 X4)))
((+ R1 X1) X3 (+ R1 X4))
((+ R1 X1) (+ R3 X3) (+ R1 X4))
((+ R1 X1) (+ R1 X4) (+ R3 (+ R2 (+ R1 X4))))
```

Consider for example the first 3-tuple of probes ((+ R1 X1) X2 (+ R1 X4)). We see that the substitution rule (1) does not apply as the random R1 occurs twice. Therefore the simulation of this 3-tuple requires the knowledge of the 3 inputs x_1, x_2 and x_4; this contradicts the t-SNI property, as only $t = 2$ intermediate variables x_2 and $y_{4,1} = x_4 \oplus r_1$ are probed, since $y_1 = x_1 \oplus r_1$ is an output variable. This shows that side-channel vulnerabilities are easy to identify in CheckMasks.

4.3. Taking into account arithmetic operations

In the description of the CheckMasks tool in [**?**], it was shown how to extend [BBD+15b] to handle a combination of arithmetic and Boolean operations, with an application to the full formal verification of the Boolean to arithmetic conversion algorithm from CHES 2017 [Cor17b]. We recall the approach below.

In order to verify the $(n-1)$-SNI property of the Boolean to arithmetic algorithm from [Cor17b], we must check that for all possible $(n-1)$-tuples of intermediate variables, the number of input variables x_i's that remain after the application of a sequence of simplification rules is always $\leq t_c$, where t_c is the number of non-output variables in the $(n-1)$-tuple.

For the verification of the RefreshMasks Boolean circuits in the previous section, we used the single simplification rule (1), namely replacing $x \oplus r$ by r when the random r appears only once in the

intermediate variables. For the verification of circuits combining Boolean and arithmetic operations, we use the following sequence of rules, where $\Psi(x, y) = x \oplus y - y$. As explained in [**?**], the ordering of the application of the rules matters, see Fig. 2.

- Rule 1: when $\omega = x_1 + x_2 \bmod 2^k$ must be simulated, simulate both x_1 and x_2.
- Rule 2: from the affine property of the function Ψ, replace $\Psi(x, y) \oplus \Psi(x, z)$ by $x \oplus \Psi(x, y \oplus z)$.
- Rule 3: from the definition of Ψ, replace $\Psi(x, y)$ by $(x \oplus y) - y \bmod 2^k$.
- Rule 4: when a random r is used only once, replace $x \oplus r$ by r, and similarly for $x + r \bmod 2^k$ and $x - r \bmod 2^k$.
- Rule 5: when a random r is not used in two intermediate variables e_1 and e_2, replace the simulation of $(e_1 \oplus r, e_2 \oplus r)$ by the simulation of $(r, (e_1 \oplus r) \oplus e_2)$; this corresponds to the change of variable $r' = e_1 \oplus r$.
- Rule 6: when $\Psi(x_1, x_2)$ must be simulated, simulate both x_1 and x_2.

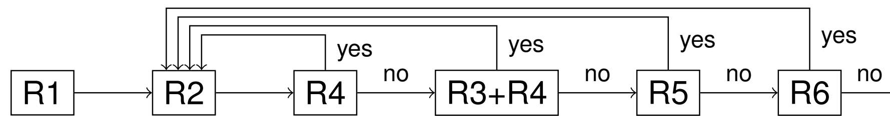

Figure 2: The rule application strategy for the formal verification of Boolean to arithmetic conversion.

n	#var.	#tuples	Security	Time
2	11	11	✓	ε
3	48	1,128	✓	0.08 s
4	133	383,306	✓	85 s
5	312	387,278,970	✓	88 h

Table 1: Formal verification of the t-SNI property of the Boolean to arithmetic conversion algorithm from [Cor17b], for $t = n - 1$.

We summarize in table 1 the timing of formal verification for the Boolean to arithmetic conversion algorithm in [Cor17b]. In summary, the `CheckMasks` tool can handle a combination of Boolean and arithmetic operation. However, handling other operations such as look-up tables (as in the table recomputation countermeasure from [Cor14]) seems more challenging.

5. The `maskVerif` approach

In this section we consider the `maskVerif` tool described in [BBD+15c, BBC+19] and publicly available at [Gré]. The `maskVerif` tool allows to describe circuits using an internal representation which is relatively flexible. It takes as input the description of Boolean circuits in a specific language, then computes the set of observations, and performs the analysis for different security properties: probing, NI and SNI security. In the following, we explain how it behaves regarding the three needs from the industry recalled in Section 3.

5.1. Program representation

maskVerif admits two input representations: a simple DSL (Domain Specific Language) allowing to describe masking algorithms, and a Verilog hardware representation. All along this subsection, we illustrate the different steps which lead to a formal verification from these two possible representations with the example of the DOM AND gadget presented in Figure 3.

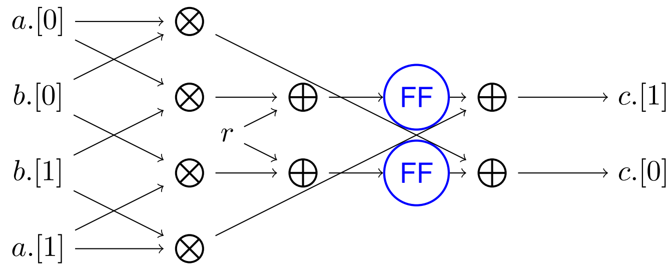

Figure 3: Graph representation of procedure DOM AND where storage in registers is indicated with blue circles

The first input representation is a simple DSL allowing to describe a masked algorithm. In this DSL, the DOM AND algorithm can be expressed as follow:

```
proc DOM_AND:
  inputs: a[0:1], b[0:1]
  (* It returns one input split in two shares *)
  outputs: c[0:1]
  (* r is a fresh random local to the algorithm *)
  randoms: r;

  a0b0 := a[0] * b[0];
  a0b1 := a[0] * b[1];
  a1b0 := a[1] * b[0];
  a1b1 := a[1] * b[1];
  aux0 = ![r + a0b1];
  aux1 = ![r + a1b0];
  c[0] := a0b0 + aux0;
  c[1] := a1b1 + aux1;
end
```

The first kind of assignment (e.g., $\mathtt{a0b0} := \mathtt{a}[0] * \mathtt{b}[0]$) will propagate glitches, while the second one (e.g., $\mathtt{aux0} =![\mathtt{r} + \mathtt{a0b1}]$) will stop the glitch propagation.

Then a simple command allows to check for the security of the algorithm with respect to different security models:

- probing security with glitches: `Probing ISW_AND`
- NI or SNI properties with glitches: `NI ISW_AND` or `SNI ISW_AND`
- probing, NI, SNI properties without glitches simply add the prefix: `noglitch`.

- by default the verification order is the number of shares of the inputs minus 1, but it can also be specified (this is particularly useful for threshold implementations):

```
noglitch order 2 Probing ISW_AND.
```

The second solution for the user is to provide a masked Verilog implementation of its algorithms and to set various parameters. Using the off-the-shelf tool (yosis), it is possible to generate an implementation in the ilang intermediate format.

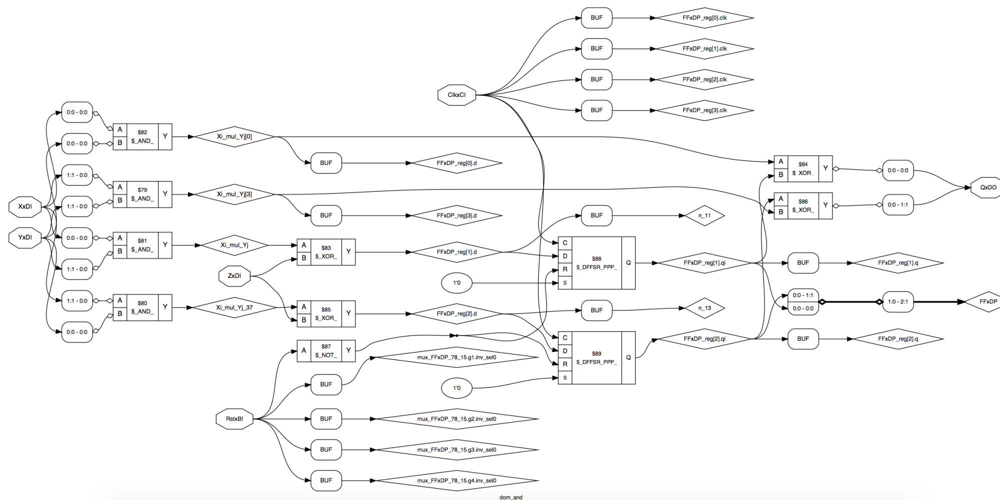

Figure 4: Diplay of DOM AND verilog implementation via Yosys

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

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

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

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

This can be captured by using the following annotations for secret inputs (or outputs):

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

Next, maskVerif generates from the annotated .ilang implementation an internal representation with a symbolic representation of leakage at each program point. At this point, verification can start as before.

For the moment, maskVerif is not able to take into account assembly implementation. The tool needs to be extended to that end.

5.2. Identification of side-channel vulnerabilities

For the different input languages of maskVerif, the tool is able determine which tuple of observation is vulnerable, and to display the error location. For example, take the following (insecure) implementation of a masked AND:

```
proc ISW_AND:
  inputs: a[0:1], b[0:1]
  outputs: c[0:1]
  randoms: r;

  c[0] := a[0] * b[0];
  a0b1 := a[0] * b[1];
  a1b0 := a[1] * b[0];
  c[1] := a[1] * b[1];
  aux  := a0b1;
  aux  := aux + a1b0;
  c[0] := c[0] ;
  c[1] := c[1] + aux;
end
```

Trying to check NI security we obtain:

```
Checking NI for ISW_AND: (no transition,no glitch)
011 tuples to check
Error ISW_AND is not NI:
Cannot check
(* stdin: line 11 (2-21) *)
(* from [a.[0] * b.[1] + a.[1] * b.[0] *)
```

which indicates that the observation line 11 can leak information.

5.3. Taking into account arithmetic operations

While the algorithms used by maskVerif are not specific to Boolean operations, maskVerif only provides Boolean operations. This is not limiting when considering Boolean circuit but this is limiting if we want to study assembly language implementations. For the need of the project, the first extension we have done is to add more base types like uint8, uint16, uint32 and uint64 with their corresponding Boolean operations.

The second extension is the possibility to add new operators. In fact the only important point for the algorithm described before is to know for each operator if it is bijective on some of its arguments. This extension allows to deal with arithmetic masking and not only with Boolean masking.

Finally, note that so far we did not implement a methodology allowing to check implementations with both boolean and arithmetic masking (in particular conversions between both systems).

Références bibliographiques

[BBC+19] 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 Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBD+15a] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, April 2015.

[BBD+15b] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 457–485, 2015. Publicly available at https://eprint.iacr.org/2015/060.

[BBD+15c] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I* [DBL], pages 457–485.

[BBD+16a] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 116–129. ACM Press, October 2016.

[BBD+16b] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016. Publicly available at https://eprint.iacr.org/2015/506.pdf. See also a preliminary version, under the title “Compositional Verification of Higher-Order Masking: Application to a Verifying Masking Compiler", publicly available at https://eprint.iacr.org/2015/506/20150527:192221.

[BIKM18] Roderick Bloem, Rinat Iusupov, Martin Krenn, and Stefan Mangard. Sharing independence & relabeling: Efficient formal verification of higher-order masking. Cryptology ePrint Archive, Report 2018/1031, 2018. https://eprint.iacr.org/2018/1031.

[BRNI13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 293–310. Springer, Heidelberg, August 2013.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 441–458, 2014.

[Cor17a] Jean-Sébastien Coron. CheckMasks: formal verification of side-channel countermeasures, 2017. Publicly available at https://github.com/coron/checkmasks.

[Cor17b] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 93–114, 2017.

[DBL] *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*. Springer.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, 2014.

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In *EUROCRYPT 2015, Part I* [DBL], pages 401–429.

[EWS14] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[Gré] Benjamin Grégoire. maskVerif. Publicly available at https://gitlab.com/benjgregoire/maskverif.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[MOPT12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 58–75. Springer, Heidelberg, September 2012.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.

[RP10a] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Heidelberg, August 2010.

[RP10b] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, pages 413–427, 2010.

[ZGSW18] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. Scinfer: Refinement-based verification of software countermeasures against side-channel attacks. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 157–177, 2018.