EPiC Series in Computing

Volume 49, 2017, Pages 17–32

PROOFS 2017. 6th International Workshop on Security Proofs for Embedded Systems

Symbolic Approach for Side-Channel Resistance Analysis of Masked Assembly Codes

Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
{ines.ben-el-ouahma, quentin.meunier, karine.heydemann, emmanuelle.encrenaz}@lip6.fr

Abstract

Masking is a popular countermeasure against side-channel attacks, which randomizes secret data with random and uniform variables called masks. At software level, masking is usually added in the source code and its effectiveness needs to be verified. In this paper, we propose a symbolic method to verify side-channel robustness of masked programs. The analysis is performed at the assembly level since compilation and optimisations may alter the added protections. Our proposed method aims to verify that intermediate computations are statistically independent from secret variables using defined distribution inference rules. We verify the first round of a masked AES in 22s and show that some secure algorithms or source codes are not leakage-free in their assembly implementations.

1 Introduction

Given the continuous growth of the Internet of Things, more and more connected devices collect, process and exchange all types of information. Personal, sensitive or valuable data are protected with cryptographic algorithms, yet the small size of IoT devices, and thus easy accessibility, has raised a new security challenge. In fact, embedded devices have been targets of a particular attack domain called physical attacks. Among them, *side channel attacks* (SCA), introduced by Kocher *et al.* in 1996 [17], aim to retrieve information about sensitive variables by statistically analysing measured physical characteristics such as power consumption or electromagnetic emanations of the circuit, closely related to the sensitive data manipulated during the execution.

These powerful attacks represent a serious threat for the security of embedded devices [22] [25]. Hence, several mitigation techniques have been proposed in literature in hardware and in software. *Masking* is a classical countermeasure which splits a secret x in d+1 shares using random variables called *masks*, so that the combination of all the shares is x [19]. The idea is to process the randomized shares instead of the secrets in order to remove the correlation of intermediate computations with the latter. A d+1 order masking protects against SCA of d-th order where an adversary can measure simultaneously d points of the executing program. Masking schemes differ according to the operation type : for example, boolean masking consists in XOR-ing a secret variable with a mask, whereas arithmetic masking adds a mask to the secret

mod 2^n. Furthermore, masking a linear function f consists in processing f separately on each share, and combining the results according to the masking scheme in order to get the expected result. However, masking a non-linear function requires the definition of another function g, which is usually done manually. This is error-prone and requires a dedicated analysis to ensure the effectiveness of the protection. At software level, masking is usually added in the source code as identification of secret variables is straightforward. However, compilation flow and optimisations can alter the masking quality. Hence, it is mandatory to ensure at assembly level that a masked program is really leakage-free. Furthermore such level for the robustness analysis is more accurate because it can take into account leakages at different architectural points such as registers, bus or ALU outputs.

The effectiveness of added countermeasures can be validated experimentally [23, 7, 1, 18, 14]. However, this approach requires specific equipment for acquiring side channel traces, which can be time consuming and error prone. Moreover, any change in the masking of a program needs a new trace acquisition. Automatic tools for generating traces have been proposed [26, 20]. Still, the trace-based evaluation uses statistical metrics which are by definition approximate and may omit to detect a leakage [24]. Another verification approach is based on formal methods which can offer stronger guaranties. For instance, [3] analyses codes in intermediate representation, which is quite close to the assembly level, but its security notion has been shown insufficient [12]. However, [12, 2] analyse source code and algorithmic levels, while they still require compilation that may invalidate the robustness evaluation results after the compilation optimisations. Therefore, there is a need for a tool able to analyse the side channel resistance of assembly or binary programs.

Our contributions. In this paper, we propose a formal symbolic approach for verifying side channel robustness of assembly programs either compiled from source code with masking countermeasures or hand-written. We focus on first order masked programs and aim to verify in the value based model that intermediate computations are statistically independent from involved secret variables. The proposed method is based on an inference system to characterise in a symbolic approach the distribution of the value of intermediate computations. We compare experimental results of our method with two enumerative techniques. The first one is a SMT-based technique that extends Eldib et al.'s work [12] in order to consider both n-width variables and assembly code. The second one is a brute-force analysis. We test the three methods on a set of common cryptographic algorithms after compilation. We show that the SMT-based approach can precisely inform about the distribution of values but is limited to few assembly instructions manipulating a very limited number of sensitive variables and masks. The brute-force approach is more efficient but still suffers from a lack of scalability. By contrast, our approach is efficient, scalable and sound, but not complete: in fact it might sometimes report some potential leak that may not exist and thus need further investigation.

The paper is organized as follows. Section 2 presents previous works for side channel resistance verification. Section 3 details the principles and modelling of the proposed symbolic approach. Section 4 discusses experimental results. Section 5 concludes and suggests directions for future work.

2 Related work

As mentionned in section 1, there are mainly two approaches to evaluate side-channel resistance: the first one is based on experimental tests using statistical tools and the second one relies on formal proofs.

A common evaluation approach is to perform a practical side-channel attack on selected program points considering a leakage model [23, 5, 7]. However, this technique is laborious and cannot cover the entire code as it is time consuming. Veshchikov et al. [26] have proposed SILK, a simulator for generating leakage traces of C++ codes for microcontrollers and microprocessors. Also, Reparaz et al. [20] have described an automatic method to generate simulated power traces considering C codes. Yet, compilation may apply instruction reordering, replacement or removal, which are common and necessary as soon as performance (both in computation time and memory footprint) is a concern. Another example is the ASCOLD tool for AVR assembly codes. It relies on the trace leakage detection tests as well to find vulnerabilities related to a specific AVR microcontroller architecture. Once leakage traces are available, *leakage detection* using the Welsh t-test is often used to analyse the side channel resistance of protected implementations [23, 7, 1, 18]. These leakage detection tests aim to detect the influence of secret variables on measured side channel traces using statistical hypothesis [14]. However, as recently pointed, it must be used carefully because it may not be able to detect some leakage [24].

The second evaluation approach relies on formal proofs [8, 15]. For instance, Coron *et al.* have proposed an algorithm for switching from arithmetic to boolean masking which is theoretically proved secure [9]. However, as mentioned previously, compilation does not guarantee identical operations and order in the generated code. We will show in the experimental results of our tool that this algorithm appears to contain some flaws after compilation.

Bayrak *et al.* have proposed a formal verification approach, implemented in the tool *Sleuth*, for analysing masked programs at LLVM intermediate representation format [3]. This approach analyses every instruction in the program according to its dependencies on program variables. An operation depending on at least one secret variable and no random variable is considered vulnerable and called a *sensitive* operation. The analysis of every operation is expressed as a SAT problem given to a SAT solver in order to assess resistance or to find out a counter-example. The *sensitivity* notion has been shown to be insufficient by Eldib *et al.* [12]. These authors have proposed an automatic formal verification tool for source codes named *SC sniffer* [12]. They consider a stronger security notion called *perfect masking* and formerly defined by Blömer *et al.* [6]. An operation involving a secret is considered *perfectly masked* if its result does not statistically depend on this secret. Eldib et al.'s approach considers boolean programs with boolean variables: given a boolean intermediate computation or a function $I(x, k, r)$ where x is plaintext, k a secret and r a random variable, this function is perfectly masked if the number of assignments of I to 1 is the same for each value of k. This property is formalized as a SMT problem with enumeration along all random variables. However, this method requires to bit-blast C programs, and to our knowledge there is no automatic tool for this transformation. Moreover, compilation may compromise the perfect masking in the target code, and therefore invalidate the result of the analysis performed at source code level. On the other hand, this approach allows to quantify the amount of information leakage [13].

Barthe *et al.* [2] propose a formal method for verifying higher-order masking schemes, based on the notion of *t-non-interference* which requires that the joint probability distribution of any t intermediate expressions is independent from secrets. Their logic is mainly based on the fact that some operators are invertible (e.g. XOR). For instance, the expression `e = k` $\oplus$ `m`, where `k` is a secret and `m` a random mask, is invertible in `m`. Considering this, their method replaces `k` $\oplus$ `m` with `m`, which implies that `e` contains no more secret deterministic variables. Consequently, the verification has proved the probabilistic non-interference of `e`. If no such invertible operator is found, the expression is simplified using algebraic normalizations in order to potentially be able to apply the rule of invertible functions. The simplification is applied only once as it may be costly. They also propose an interesting method for building large sets

of intermediate variables for high order verifications but we do not discuss this as our work focuses on first order masking countermeasures. In summary, their approach is appealing and gives impressive results.

The approach we proposed in this article also works on secret independence, like previously cited works, but uses finer grain rules to determine statistical independence w.r.t. a secret. In fact, [2] consider only the probabilistic and deterministic nature of variables, while we define more precise distribution types. Moreover, our method applies to boolean and arithmetic operations, while their proof system relies mainly on the existence of invertible functions.

3 Symbolic distribution method

In this section, we present our method for the verification of statistical independence of intermediate computations from secret variables. This method relies on a symbolic approach to characterize the distribution of the values of an expression. The characterization consists of a type, which specifies either that the distribution is uniform or that it is statistically independent from secrets. The goal is to be able to infer, as much as possible, uniform expressions or else statistically independent from secrets expressions, by combining sub-expressions, depending of the operator type and the sub-expressions dependencies. To achieve this goal, inference rules have been defined for different operators, with the resulting types and dependencies. The starting point is to tag all input variables with the corresponding distribution: uniform for masks and unknown for secrets. Then, an analysis of the expression tree is performed by type inference of intermediate nodes, until the root is reached. The tag for this node characterises the distribution of the global expression result.

In the following, we first define the scheme of verification. Then, we define the distribution types and the independence notions. Next, we explain the main inference rules and give an overview of the inference mechanism. Finally, we present a lower granularity analysis for handling some special cases.

3.1 Verification Scheme

Our approach focuses on implementations of cryptographic algorithms with masking countermeasures. It considers assembly codes composed of common operators that appear in such programs like boolean, arithmetic, shift and rotation operations. They may also contain conditional statements, bounded loops and memory accesses. However, the control-flow of the program to analyse must be statically evaluable; in particular, there must be no indirect jumps and the number of iterations of loops must be known at compile time. Cryptographic algorithms usually fulfill this requirement. If this number cannot be statically determined, then the loop break condition is input-dependent which implies that we cannot know when the loop ends since our analysis is performed on symbolic variables and not on values. In future work, we can consider to set a maximum number of iterations analysed for such loops.

The principle is to analyse the expression of the destination register content for each instruction in the program. User annotations must tag registers containing input variables or memory locations as either secret variable or random uniform mask. Memory store instructions are ignored because the register content being written in memory has necessarily been already analysed as the result of a previous instruction. Every memory load is supposed to read initialized data, which is either tagged at the beginning, or which is a result of a previously stored and thus analysed expression.

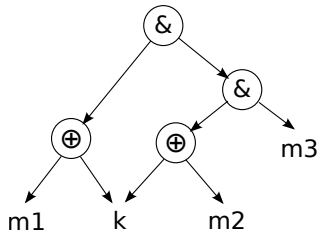
We verify programs in the *value-based* leakage model in which the leakage is related to the computed values, which differs from the *transition-based* model where it is the differential between the content of a register and the new written value that leaks. A program is *1st* order side-channel free if the distribution of all intermediate computation results is statistically independent from involved secret variables. The verification process follows the steps below:

- A *data dependency graph* (DDG) is computed for **each instruction** i of the program under test and corresponds to the resulting expression of i. In this graph, leaves are input variables either loaded from memory or initially stored in registers, that can be either secret or not. Intermediate nodes correspond to operations of intermediate (previous) instructions and the root of the graph corresponds to the operation of i.
- The DDG corresponding to an instruction is then analysed independently from other instructions DDGs in order to check that its distribution is statistically independent from the secrets appearing in it. This analysis can use simplification rules such as laws of boolean algebra. For example, the expression `(k ⊕ m ⊕ m ⊕ 0)` is simplified as `k`. This simplification phase has no influence on the distribution of the instruction expression and the original and simplified expressions are functionally equivalent. In particular, shift and rotation operations are transformed into a combination of bit extractions and concatenations.

Consider the assembly code example shown in Figure 1. The registers `r0`, `r1`, `r2` and `r3` contain respectively a secret variable `k` and masks `m1`, `m2`, `m3`. Figure 1b shows the data dependency graph of the last instruction which is associated to the expression `(k ⊕ m1) & ((k ⊕ m2) & m3)`. The verification is applied to every instruction, hence intermediate expressions are analysed as well.

```
# r0 <- k; r1 <- m1; r2 <- m2;
# r3 <- m3
eor r4, r0, r1   # k ⊕ m1
eor r5, r0, r2   # k ⊕ m2
and r5, r5, r3   # (k ⊕ m2) & m3
and r5, r5, r4
# (k ⊕ m1) & ((k ⊕ m2) & m3)
```

(a) Assembly code



(b) DDG of the last instruction

Figure 1: Assembly code example and DDG of the last instruction. The DDGs of the previous instructions correspond to subtrees of this DDG

3.2 Distribution types

For a program, we distinguish 2 sets of variables: S the set of secret variables, and M the set of masks. We call E the union set of S and M. Masks in M are supposed to be uniform and independent. Masking actually relies on the existence of such an ability to generate uniform and independent masks. However no assumption is made upon the distribution of variables in S. These variables can be stored in registers or loaded from memory.

Definition 1. We define the following 4 distribution types for variables and expressions:

- Random Uniform Distribution (RUD): Let x be a variable or an expression defined on n bits, x is tagged RUD if and only if P(x = c) = $\frac{1}{2^n}$ $\forall$ c a constant $\in$ [0;2^n-1].
- Unknown Distribution (UKD)
- Constant (CST)
- (Statistically) Independent of Secrets Distribution (ISD): a distribution which is not necessarily uniform but is identical for all values of the secrets. Note that a RUD is necessarily an ISD. For example, let e = (k $\oplus$ m_1) & m_2 defined on 1 bit with k a secret and m_1, m_2 masks. By enumeration over the 3 variables, we obtain the following probabilities: $P(e = 0 \mid k = 0) = P(e = 0 \mid k = 1) = \frac{3}{4}$ and $P(e = 1 \mid k = 0) = P(e = 1 \mid k = 1) = \frac{1}{4}$. Therefore, e is ISD, while it is not RUD.

Independence notions. We distinguish 2 notions of dependency for an expression e w.r.t a variable v. If v appears in the expression of e, then e is said to be *structurally* depending on v. If the distribution of the result of e depends on v, then e is *statistically* depending on v. Therefore, 2 notions of independence arise: we consider an expression e to be structurally independent of a variable v if v doesn't appear in the expression of e. Conversely, we say that an expression e is statistically independent of v if its distribution is independent of the value of v, but v can appear in the expression of e.

For example in the common boolean masking scheme, we have the expression e = m $\oplus$ k where k, a secret variable tagged UKD, is masked with a mask m, tagged RUD (m and k are independent and on n bits). Here, e is statistically independent of k but structurally depending on it, as will be shown in section 3.3.

During the analysis, we keep track of structural dependencies because they may invalidate a masking. For example consider the expression e = ((k $\oplus$ m) & m) on 1 bit, where k is a secret so it is tagged UKD and m is a mask so it is tagged RUD. Given the property of XOR, the sub-expression (k $\oplus$ m) is tagged RUD. If we analyse the whole expression as (RUD & RUD), it yields ISD which means that e is statistically independent of k. However, if we compute the distribution of e, we obtain that $P(e = 0 \mid k = 0) = \frac{1}{2}$, while $P(e = 0 \mid k = 1) = 1$. Consequently, e is in fact statistically depending on k. Hence, structural dependencies have to be kept.

Safe types. To show that a register content is statistically independent of secrets, the distribution of its expression needs to be either RUD, ISD, or UKD with no structural dependency on any variable in S. Therefore the only unsafe tag is UKD with structural dependency on some variable in S.

Notations. In the following, an expression e, whose distribution is D, is written e~D. The structural dependency of e w.r.t a set dep of variables in E is written e{dep}. For example for e = m $\oplus$ k, we write e~RUD{m,k}. Concerning variables, we write m~RUD{m} for a mask m since it depends structurally on itself by definition. Similarly, a secret k is written as k~UKD{k}.

3.3 Inference rules

In this article, we present a subset of the defined rules where we keep track only of structural dependencies and dominant masks introduced below. First we define two general rules called Dom and Disjoint.

Proposition 1. Let m~RUD{m} be a mask and e~UKD{dep} be an expression such that m $\notin$ dep. Then (m $\oplus$ e)~RUD{{m} $\cup$ dep}. In particular, we note that this distribution is statistically independent of e, since it is uniform with no assumption on the distribution of e.

Proof. Consider expressions and variables on n bits. $\forall\ \mathtt{v}_0 \in [\![0; 2^n - 1]\!]$, the probability that $\mathtt{e} \oplus \mathtt{m}$ takes a value $\mathtt{v}_0$ is:

$$\begin{aligned}
P(e \oplus m = v_0) &= \sum_{e_0 \in [\![0;2^n-1]\!]} P(e \oplus m = v_0 | e = e_0) \cdot P(e = e_0) \\
&= \sum_{e_0 \in [\![0;2^n-1]\!]} P(e_0 \oplus m = v_0 | e = e_0) \cdot P(e = e_0) \\
&= \sum_{e_0 \in [\![0;2^n-1]\!]} P(m = e_0 \oplus v_0) \cdot P(e = e_0) \text{ because } \mathtt{m} \text{ and } \mathtt{e} \text{ are independent} \\
&= \sum_{e_0 \in [\![0;2^n-1]\!]} \frac{1}{2^n} \cdot P(e = e_0) \text{ because } \mathtt{m} \text{ is uniform} \\
&= \frac{1}{2^n} \cdot \sum_{e_0 \in [\![0;2^n-1]\!]} P(e = e_0) \\
&= \frac{1}{2^n}
\end{aligned}$$

□

Corollary 1 (Dom)**.** Let $\mathtt{e}$ be an expression on $\mathtt{n}$ bits that can be written as $\mathtt{e = m}_0$ *op* $\mathtt{m}_1$ *op* ... *op* $\mathtt{m}_t$ *op* $\mathtt{e'}$ such that:

(i) *op* $\in \{\oplus, \mathtt{add\ mod}\ 2^n\}$,
(ii) $\forall\ \mathtt{j}, 0 \leq \mathtt{j} \leq \mathtt{t}, \mathtt{m}_j \in M$
(ii) $\mathtt{dep(e')} \cap \mathtt{m}_0 \cap \mathtt{m}_1 \cap \ldots \cap \mathtt{m}_t = \varnothing$,

then $\mathtt{e}$ is tagged RUD and $\mathtt{m}_j$ are said to be *dominant masks* of $\mathtt{e}$. Informally, if there exists a mask that XORs an arbitrary expression $\mathtt{e'}$ or is added to it mod 2^n, and if this mask does not appear in $\mathtt{e'}$, then the resulting expression $\mathtt{e}$ is random and of uniform distribution.

We keep track of dominant masks for each expression and distinguish a set for XOR dominant masks and another set for additive dominant masks. Theses sets do no appear in the presented rules in order to simplify the writing, but they are maintained as follows. If we write $\mathtt{dom}_\oplus\mathtt{(e)}$ the set of xor dominant masks of $\mathtt{e}$ and $\mathtt{dom}_+\mathtt{(e)}$ the set of additive dominant masks of $\mathtt{e}$, we have:

- $\mathtt{dom}_+\mathtt{(e0 \oplus e1) = \varnothing}$
- $\mathtt{dom}_\oplus\mathtt{(e0 \oplus e1) = dom}_\oplus\mathtt{(e0) \cup dom}_\oplus\mathtt{(e1) - (dom}_\oplus\mathtt{(e0) \cap dom}_\oplus\mathtt{(e1))}$
- $\mathtt{dom}_\oplus\mathtt{(e0 + e1) = \varnothing}$
- $\mathtt{dom}_+\mathtt{(e0 + e1) = dom}_+\mathtt{(e0) \cup dom}_+\mathtt{(e1) - (dom}_+\mathtt{(e0) \cap dom}_+\mathtt{(e1))}$

Proposition 2 (Disjoint)**.** Let $\mathtt{u} \sim \mathtt{ISD\{dep0\}}$ and $\mathtt{v} \sim \mathtt{ISD\{dep1\}}$. For every binary operation op, if $\mathtt{dep0} \cap \mathtt{dep1} \cap M = \varnothing$, then $\mathtt{(u\ op\ v)} \sim \mathtt{ISD\{dep0 \cup dep1\}}$. In other words, if the expressions $\mathtt{u}$ and $\mathtt{v}$ are ISD, and the masks in $\mathtt{u}$ do not appear in $\mathtt{v}$ and vice versa, then the resulting expression of a binary operation of $\mathtt{u}$ and $\mathtt{v}$ is ISD.

Proof. Since $\mathtt{u}$ is statistically independent from S, then each bit of each occurrence of each secret variable in $\mathtt{u}$ is masked with a variable in M (somewhere in the expression tree) so that the result is uniformly distributed. The same is true for $\mathtt{v}$. Since expressions $\mathtt{u}$ and $\mathtt{v}$ have no masks in common, all masked expressions resulting from the occurrences of all secret variables in $\mathtt{u}$ are independent from all masked expressions resulting from secret variables in $\mathtt{v}$. Thus, $\mathtt{u}$ and $\mathtt{v}$ are statistically independent for every binary operation op.

In the following, we consider only one secret variable $\mathtt{k}$, and n bit variables, without loss of generality. $\forall\ c_0 \in [\![0; 2^n - 1]\!]$, $k_0 \in [\![0; 2^n - 1]\!]$, we have

$$
\begin{aligned}
P(u\ op\ v = c_0 \wedge k = k_0) &= \sum_{u_0,v_0 \mid u_0\ op\ v_0 = c_0} P(u = u_0 \wedge v = v_0 \wedge k = k_0) \\
&= \sum_{u_0,v_0 \mid u_0\ op\ v_0 = c_0} P(u = u_0 \wedge k = k_0) \cdot P(v = v_0) && \text{because } (\texttt{u}, \texttt{k}) \text{ independent from } \texttt{v} \\
&= \sum_{u_0,v_0 \mid u_0\ op\ v_0 = c_0} P(u = u_0) \cdot P(k = k_0) \cdot P(v = v_0) && \text{because } \texttt{u} \text{ is independent from } \texttt{k} \\
&= P(k = k_0) \sum_{u_0,v_0 \mid u_0\ op\ v_0 = c_0} P(u = u_0) \cdot P(v = v_0) \\
&= P(k = k_0) \sum_{u_0,v_0 \mid u_0\ op\ v_0 = c_0} P(u = u_0 \wedge v = v_0) && \text{because } \texttt{u} \text{ and } \texttt{v} \text{ are independent} \\
&= P(k = k_0) \cdot P(u\ op\ v = c_0)
\end{aligned}
$$

□

Furthermore, we define specific inference rules for each operation. In the following, we present the set of rules for the XOR and addition mod 2^n that share the same rules, and the set of rules for AND and OR operators which have common rules as well. We omit to write the rules that produce `UKD` tags for the presented operators. The subtraction `(u - v)` is simplified as `(u + bitwise-not(v) + 1)`, therefore its analysis relies on the rules of the addition operator. Analysing the NOT operator amounts to analysing its argument.

XOR and Addition mod 2^n In the following rules, the considered dominant masks are those of the same operator, i.e. $\texttt{dom}_{\oplus}$ for the XOR, and $\texttt{dom}_{+}$ for the addition modulo 2^n.

1. `z = u~RUD{dep0} {⊕, +} v~RUD{dep1}`
   - `z~RUD{dep0 ∪ dep1}` if $\exists$ `m` $\in M$, such that (`m` $\in$ `dom(u)` and `m` $\notin$ `dep1`) or (`m` $\in$ `dom(v)` and `m` $\notin$ `dep0`)
     Following the corollary 1, we look for a dominant mask in `u` or in `v` that can render `z RUD`.
2. `z = u~RUD{dep0} {⊕, +} v~ISD{dep1}`
   - `z~RUD{dep0 ∪ dep1}` if $\exists$ `m` $\in M$ such that `m` $\in$ `dom(u)` and `m` $\notin$ `dep1`
3. `z = u~RUD{dep0} {⊕, +} v~UKD{dep1}`
   - `z~RUD{dep0 ∪ dep1}` if $\exists$ `m` $\in M$ such that `m` $\in$ `dom(u)` and `m` $\notin$ `dep1`
4. `z = u~RUD{dep} {⊕, +} v~CST`
   - `z~RUD{dep}`
5. `z = u~ISD{dep0} {⊕, +} v~ISD{dep1}`
   - `z~ISD{dep0 ∪ dep1}` if `dep0` $\cap$ `dep1` $\cap M = \varnothing$ (cf. Rule Disjoint)

AND, OR

1. `z = u~RUD{dep0} {&, |} v~RUD{dep1}`
   - `z~ISD{dep0 ∪ dep1}` if `dep0` $\cap$ `dep1` $\cap M = \varnothing$
   - `z~ISD{dep0 ∪ dep1}` if $\exists$ `m0` such that `m0` $\in$ `dom(u)` and `m0` $\notin$ `dep1`, or if $\exists$ `m1` such that `m1` $\in$ `dom(v)` and `m1` $\notin$ `dep0`

2. `z = u~RUD{dep0} {&, |} v~ISD{dep1}`
   - `z~ISD{dep0 ∪ dep1}` if `dep0 ∩ dep1 ∩ M = ∅` (cf. rule Disjoint))
3. `z = u~RUD{dep} {&, |} v~CST`
   - `z~ISD{dep}`
4. `z = u~ISD{dep0} {&, |} v~ISD{dep1}`
   - `z~ISD{dep0 ∪ dep1}` if `dep0 ∩ dep1 ∩ M = ∅` (cf. rule Disjoint)

Inference mechanism. Algorithm 1 describes the analysis performed recursively on a DDG, starting from the root operation of the instruction. It tries to infer the `RUD` type, then the `ISD` type. If no rule is found, an `UKD` type is raised, meaning that no specific information on the distribution and statistical dependencies is known.

Algorithm 1 Distribution inference algorithm

```
1:  function infer(e)
2:      if e is a leaf then
3:          if e ∈ S then return UKD{e}
4:          else if e ∈ M then return RUD{e}
5:          else return CST
6:      else
7:          le{ld} = infer(e.left_child)
8:          re{rd} = infer(e.right_child)
9:          if ∃ rule for (le{ld} e.op re{rd}) that returns RUD{dep} then
10:             return RUD{dep}
11:         else if ∃ rule for (le{ld} e.op re{rd}) that returns ISD{dep} then
12:             return ISD{dep}
13:         else return UKD{dep}
```

Example. We use our running example to illustrate the inference mechanism. Figure 2 shows the resulting inference tags for the expression `(k ⊕ m1) & ((k ⊕ m2) & m3)` corresponding to the last instruction in the example in figure 1. The root of the DDG is tagged `ISD` so the result of this instruction is statistically independent from the secret `k`.

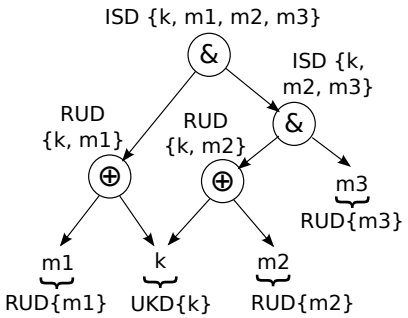


Figure 2: Distribution inference of the running example

3.4 Bit level analysis

For cases where the distribution analysis fails to prove independency from the secrets, a finer grain strategy is used, in which the resulting expression is split into several expressions, with a bit granularity: for each bit of the resulting expression, the corresponding dependency graph is created and analysed. If all these expressions are RUD, a dominant mask is searched which does not appear in the other sub-expressions (at the bit granularity). In that case, this means that all the bit expressions are independent and that the global result is RUD. If this not the case, we try to check if the result is ISD, using the fact that the concatenation of an ISD bit with a CST expression and the deduplication of an ISD bit both produce an ISD result. If this strategy can work well with shift and bitwise operators, it is more costly and does not necessarily give good results with arithmetic operators, since the carry computation is more complex and typically mixes the mask bits ranks. Yet, this strategy allows to deal with expressions in the form of bit-fields concatenations (e.g. `(e0 & 0xF0) | (e1 & 0xF)`).

4 Experiments

In this section, we present experimental results of our method on a set of first order masked assembly programs described in literature. We implemented our method in a tool that analyses ARM Thumb-2 assembly codes. Widely used in embedded devices, ARM processors have a 32-bit instruction set with 16 general registers numbered from 0 to 15. The Thumb mode allows to mix 16-bit and 32-bit instructions to reduce the code size, which is useful in resource constrained devices. To model a program, we used z3py, the Python binding of Z3 (a SMT solver) [21]. The Python binding facilitates the writing of expressions thanks to the accessibility to Python structures (loops, if-else, assignments...). Therefore, we implemented our distribution inference algorithm in Python, using the data structures of z3py for representing variables of the assembly programs; however, we limited the use of z3py functions to the `simplify` function, which performs local simplifications in expressions.

As said in Section 1, we compare our method with two enumeration-based techniques which are described in the following. Afterwards, we present the selected test examples and provide experimental results for our method and the two compared methods. We discuss these results and expose the advantages and limitations of our method that we compare with the enumeration-based methods scores. Finally, we conclude by proposing a possible extension of our method for higher order masked programs and for the transition-based leakage model.

4.1 Enumeration techniques

We implemented two enumeration techniques. The first one, called *C-enumerative* generates a C program in which the result of the expression is computed for all possible combinations of variable values. Then, the expression distributions are compared according to the secret values, showing the distribution type. The second enumeration technique, called *SMT-enumerative*, is based on Eldib's SMT technique [12] presented in Section 2. The SMT formulation enumerates explicitly on values of masks and implicitly (i.e. via the solver) on values of secret variables. Finally it asks the solver to find two different values of a secret variable for which the resulting expression gives different distributions. In the *SMT-enumerative* method, we extended Eldib's approach to deal with multi-bit variables. The SMT problems are generated in the SMT-lib standard format in order to be checked by different solvers if desired. The *C-enumerative* method can serve as a reference for resulting distributions, and return whether an expression

Table 1: Benchmark characteristics

Program	#ASM inst	Size in bits	#Masks	#Secrets	#Vulnerabilities in literature
P6 [12]	8	1	3	3	2
Masked Chi [12]	8	1	2	3	0
Goubin Conversion [15]	8	4	2	1	0
Coron Conversion [9]	37	4	3	1	0
Masked AES 1st round [16]	422	8	6	16 + 16	0
Simon TI 1st round [23]	15	32	5	3 + 2	0

is vulnerable or if it is RUD or ISD following the tag definitions of the symbolic approach. The *SMT-enumerative* technique only gives statistical dependence (or independence) w.r.t. the secrets, which is enough for tracking our leakage model.

4.2 Benchmarks

Table 1 presents the characteristics of the test suite we considered. The first column lists the program names and their references. The second and third columns show respectively the number of assembly instructions and the data width. Next, columns 4 and 5 give the number of input masks and secret variables of each program. We consider plain texts encrypted by a cipher as secrets, like in [10]. The last column contains the number of vulnerabilities for each program according to their description in references at source or algorithmic level.

Our benchmark starts with two small boolean programs: P6 [1] which is a logic design of the AES S-Box from the test suite in [12], and a masked C version of a bit blasted Chi function synthesised in [11]. Subsequently, we considered an algorithm of Goubin *et al.* [15] that converts a boolean masking to an arithmetic masking, and an algorithm of Coron *et al.* [9] which performs the reverse. The latter contains a loop for which the number of iterations increases with the input size, and for simplicity we considered 4-bit data requiring a single iteration. Goubin and Coron algorithms are designed for functions combining different types of operations, for instance boolean and arithmetic, which require different types of masking. Next, the AES being a state of the art algorithm, we implemented a masked version of 128-bit AES according to the masking scheme in [16]. In this scheme, 6 masks are used to harden the entire algorithm with no mask refresh between rounds. Finally, the Simon block cipher for 32-bit words ends the benchmarks. Introduced in 2013 [4], it is a family of lightweight block ciphers optimized for hardware implementations that offers a trade off between size and security level for constrained hardware and software environments. We implemented a masked version of this block cipher as explained in [23] which is based on the Threshold Implementation (TI) protection with two shares. All benchmarks implementations were done in C and compiled to the targeted ARM assembly with GCC and the optimisation level -O2, commonly used for combined performance and size optimisations.

[1] http://www-bcf.usc.edu/~wang626/project_sniffer.htm

4.3 Results

Table 2 draws a parallel between analysis results of our method and those of the enumeration-based methods. For the symbolic distribution method, it gives the number of instructions tagged with each of the 4 defined distribution types presented in 3.2. For the *C-enumerative* technique, it shows the number of instruction results which are RUD, ISD or which are vulnerable, i.e. for which there exist two different secret values that produce different distributions for the expression. The *SMT-enumerative* detects the same vulnerable instructions as the *C-enumerative*, however it cannot distinguish between ISD and RUD.

The assembly code of P6 program contains instructions which all have independent distributions from secrets, while its source code is leaky [12]. The masked Chi C implementation has been shown to have all its intermediate computations statistically independent from secrets [12]. However, the corresponding assembly code appears to have 4 vulnerable instructions using enumerative techniques, which are the same 4 instructions typed as unsafe by the symbolic distribution method. This is due to instruction reordering or computation simplification performed by the compilation phase. These vulnerabilities were not present in table 1 and illustrate the necessity to verify the efficiency of added masking protections after compilation.

In [15], each computation of the Goubin Conversion algorithm is shown to be independent from the secret, thus it is proven secure. According to enumerative methods, the generated assembly program turns out to have no vulnerable instructions either. However, the symbolic method gives 3 instructions tagged `UKD`, thus it cannot say anything about their distributions. Similarly, [9] demonstrates that each intermediate variable in the Coron Conversion algorithm is uniformly distributed. Yet, the assembly program presents 7 vulnerable instructions according to the enumeration techniques. These 7 instructions are included in the 13 instructions tagged unsafe by the symbolic method. Again, we observe the need to check side channel resistance at assembly level since these leakages result from some compiler reordering or optimization. Also, these examples illustrate the incompleteness of our method which cannot demonstrate statistical independence from secrets for 6 instructions, while they are leakage-free according to the enumeration methods.

The masked AES algorithm is proved secure against first order side channel attacks in [16]. The generated assembly implementation cannot be verified by the enumeration techniques because of the number of input variables and their size in bytes. On the other hand, our symbolic distribution method can perform an analysis of the first round in a reasonable amount of time (less than a minute, cf. Table 3). In this round, all instructions are tagged with safe tags.

Simon TI algorithm with two shares does not meet the TI requirements for hardware implementations, but is said to be suited for software implementations if an order process can be forced [23]. In our C implementation, we introduced parenthesis to specify the requested computation order. However, the computation order imposed by added parenthesis was not respected by the compiler which applied reordering based on the commutative and associative laws of XOR; the generated code produced a leaking intermediate result. As in the case of AES, Simon TI program cannot be analysed by the enumerative methods. Concerning the symbolic distribution analysis, 3 unsafe instructions were found in the first round and 2 in the next five rounds. The expression of one of the 3 potentially vulnerable instructions is: `ROL(t0,8)` $\oplus$ `ROL(m2,8)` $\oplus$ `ROL(m2,8)`, in which `ROL` stands for a left rotation, `t0` is one of the two plain text words and `m2` a mask. In fact, this expression is reducible to `ROL(t0,8)`, hence the plain text is revealed, while it must be masked as considered secret.

Table 4 shows the number of types obtained with the symbolic method at word level and at bit granularity. For instance, the column #`RUD`$_w$ gives the number of instructions tagged as

Table 2: Analysis results

Program	Ref (enumeration)			Symbolic			
	# RUD	# ISD	# Vuln	# RUD	# ISD	# UKD	# CST
P6	6	2	0	6	2	0	0
Masked Chi	2	2	4	2	2	4	0
Goubin Conversion	7	1	0	5	0	3	0
Coron Conversion	19	11	7	14	10	13	0
Masked AES 1st round	-	-	-	302	0	0	120
Simon TI 1st round	-	-	-	7	4	3	1

Table 3: Verification time

Program	Symbolic # indep	Enum C/SMT # indep	Total indep	Symbolic time	Enum C time	SMT time
P6	8	8	8	<1s	<1s	<1s
Masked Chi	4	4	4	<1s	<1s	<1s
Goubin Conversion	8	8	8	<1s	<1s	35mn
Coron Conversion	24	30	30	2s	1s	5,6h
Masked AES 1st round	422	-	422	22s	-	-
Simon TI 1st round	13	-	13	8.5s	-	-

RUD after the word analysis, while the column #$\texttt{RUD}_b$ gives the number of instructions that were first tagged UKD and which resulted RUD after the analysis at bit level. The columns #<tag> total give the final number of instructions tagged as <tag>. We notice that for the Coron conversion and Simon programs, the bit level analysis allows to give safe types for around 40% of the instructions initially tagged as unsafe. In the case of the AES, it even removes all unsafe tags. Although it requires to breakdown an expression and performs more analyses, the bit level analysis appears to be very useful as it can render safe a relatively considerable percentage of initially unsafe instructions.

Scalability and accuracy. Both enumerative methods may not terminate due to the combinatorial blow-up of the enumeration over all variables of an expression. Eldib *et al.* analyse bit-blasted codes with their SMT tool; however this transformation is not trivial, and to the best of our knowledge there is no tool for bit-blasting programs. There is clear evidence that this aspect decreases the scalability of both enumerative methods, as shown in the experiments. On the other hand, the distribution method is independent from the data size and is linear in the number of variables and operations of the analysed expression. When the bit granularity is needed, each bit is analysed separately, which multiplies the required computations by the number of bits.

Quite evidently, the *C-enumerative* and *SMT-enumerative* methods are sound. The sym-

Table 4: Breakdown results for the symbolic method

Program	#RUD$_w$	#RUD$_b$	#RUD total	#ISD$_w$	#ISD$_b$	#ISD total	#UKD$_w$	#UKD$_b$	#UKD total	#CST	#total inst
P6	6	0	6	2	0	2	0	0	0	0	8
Masked Chi	2	0	2	2	0	2	4	4	4	0	8
Goubin Conv.	0	0	5	0	0	0	3	3	3	0	8
Coron Conv.	10	4	14	6	4	10	21	13	13	0	37
Masked AES 1st round	222	80	302	0	0	0	80	0	0	120	422
Simon 1st round	7	0	7	0	3	3	7	4	4	1	15

bolic distribution method is sound as well, since the inference system guaranties the statistical independence from secrets when stated. However, only the enumerative methods are complete (when they terminate). Indeed, the symbolic method can tag an expression as UKD with structural dependencies on some secret (i.e. unsafe), while in fact its distribution does not truly depend on this secret.

Compared to Barthe *et al.* [2], we think that our method can analyse finer expressions, such as (k ⊕ m1 ⊕ m2) & (k ⊕ m2), for which it can conclude ISD whereas from our understanding, their algorithm cannot conclude. Besides, the bit-level analysis with bit-field concatenations and constant propagation is another case easily processed with our method for which [2] seems not to be very adapted.

Masking order and transition-based leakage model. We presented our method for the analysis of first order side channel robustness in the value based model. The presented method is not directly applicable to higher order masking. However, it can be extended to analyse such programs using the following approach. Once the analysis of all expressions at the first order is complete, all the tuples of values (pairs at the second order, triples at the third order, etc.) need to be checked to ensure that no combination can reveal dependence with the secret. At the second order, this can be done for instance by checking that for each pair, one of the expressions of the pair has a dominant mask which does not appear in the structural dependencies of the other expression. More generally, at order k, considering a set of expressions $(e_0, ..., e_{k-1})$, there must be an expression e_p such that a dominant mask in e_p does not appear in the structural dependencies of any of the $e_i (i \neq p)$, and such that $(e_0, ..., e_{p-1}, e_{p+1}, ..., e_{k-1})$ does not leak information at the order $k-1$.

Another leakage model is the transition-based model. In this model, the value leaked by an instruction is a combination of the result of the instruction and the previous value contained in the destination register. In the more general framework, we can consider that both values are entirely leaked. The analysis in a first order transition-based model is then equivalent to an analysis in a second-order value-based model (if we consider initial register values to be irrelevant). A less general framework is to consider that the value leaked is the result of the XOR of both the previous and the new values of the register, following the idea that the leak

is highly dependent on the modified bits. In this case, the presented method can be adapted by computing the XOR of the current instruction with the value previously contained in the destination register before performing the expression analysis. Of course, this approach requires more computations, since expressions are globally twice larger.

5 Conclusion and perspectives

We proposed a symbolic method for verifying side channel robustness of first-order masked programs in the value-based model. The analysis relies on an inference system which characterises the distribution of an expression according to defined distribution types. An expression is leakage-free if its distribution is statistically independent from all the secrets appearing in it. We implemented our method for the analysis of programs at the assembly level in order to take into consideration the compilation process. We compared this symbolic method with two enumerative approaches, one in C and one using a SMT formulation, and showed that the symbolic method scales much better when the size of the expression grows. For instance, the symbolic method was able to verify a full round of the AES, while the enumerative methods quickly reach their limits in computation time. Besides, the symbolic approach allowed to reveal two flaws in a masked Simon implementation which was supposed to be safe at the software level, due to compilation optimisations. Yet, the drawback of the symbolic method is that it is not complete, meaning that in some cases, instructions cannot be tagged as safe while in fact they are.

Future work include refinement of the set of rules to reduce the number of unsafe tags making our method incomplete, and eventually define cases where the leakage can be stated with certainty, which avoids the need for further analysis of the given expression. The refinement of the bit level analysis can improve the results accuracy as well. In cases where the bit level analysis fails to state the statistical independence, we can consider using the enumerative approach at the bit level but need to take into account inter-bit dependencies. As discussed in the results section, another line of research would be the extension of the symbolic approach to consider other leakage models or to analyse the side channel resistance of higher masking orders.

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, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Eurocrypt 2015*, number 9056, 2015.

[3] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic application of power analysis countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.

[4] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.

[5] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In *Lecture Notes in Computer Science*, volume 8874, pages 326–343. Springer-Verlag, 2014.

[6] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. In *International Workshop on Selected Areas in Cryptography*, pages 69–83. Springer, 2004.

[7] Cong Chen, Mehmet Sinan Inci, Mostafa Taha, and Thomas Eisenbarth. Spectre: A tiny side-channel resistant speck core for fpgas. *IACR Cryptology ePrint Archive*, 2015:691, 2015.

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

[9] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *International Workshop on Fast Software Encryption*, pages 130–149. Springer, 2015.

[10] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side channel cryptanalysis of a higher order masking scheme. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 28–44. Springer, 2007.

[11] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *CAV*, volume 8559, pages 114–130, 2014.

[12] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):11, 2014.

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

[14] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, pages 158–172, 2011.

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

[16] 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.

[17] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[18] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. *IACR Cryptology ePrint Archive*, 2017:345, 2017.

[19] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 142–159. Springer, 2013.

[20] Oscar Reparaz. Detecting flawed masking schemes with leakage detection tests. In *Lecture Notes in Computer Science*. Springer-Verlag, 2016.

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

[22] Eyal Ronen, Colin OFlynn, Adi Shamir, and Achi-Or Weingarten. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. Technical Report 1047, 2016.

[23] Aria Shahverdi, Mostafa Taha, and Thomas Eisenbarth. Lightweight side channel resistance: Threshold implementations of simon. *IEEE Transactions on Computers*, 66(4):661–671, 2017.

[24] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. *IACR Cryptology ePrint Archive*, 2017:138, 2017.

[25] D. Strobel, D. Oswald, B. Richter, F. Schellenberg, and C. Paar. Microcontrollers as (in)security devices for pervasive computing applications. *Proceedings of the IEEE*, 102(8):1157–1173, Aug 2014.

[26] Nikita Veshchikov. Silk: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 3. ACM, 2014.