Maskara: Compilation of a Masking Countermeasure with Optimised Polynomial Interpolation

Nicolas Belleville, Damien Couroussé, Karine Heydemann, Quentin Meunier, Inès Ben El Ouama

Abstract—**Side-channel attacks are amongst the major threats for embedded systems and IoT devices. Masking is one of the most used countermeasure against such attacks, but its application remains a difficult process. We propose a target-independent approach for applying a first-order boolean masking countermeasure during compilation, on the static single assignment form. Contrary to state-of-the art automated approaches that require to simplify the control flow of the input program, our approach supports regular control-flow program structures. Moreover, our compiler is the first to automatically mask table lookups using a polynomial interpolation approach. We also present new optimisations to speed up the evaluation of polynomials: we reduce the number of terms of the polynomial, and we accelerate finite field multiplication. We show that our approach is faster than the standard masked table approach with mask refresh after each access, with speedups up to $\times 2.4$ in our experiments. Finally, using a formal verification approach, we show that the compiled machine code is secure, i.e., that all intermediate computations are statistically independent of the secrets.**

Index Terms—**side-channel attack, masking, compiler**

I. Introduction

Side-channel attacks aim at recovering a secret manipulated in a numerical computation by exploiting measurements of physical quantities such as the power consumption or the electromagnetic emissions [?]. Side-channel attacks are famous for their ability to recover cryptographic keys, even though they can be used as well to recover different secrets, or to reverse-engineer a program executed. These attacks can be applied to all kinds of computing systems, but they are particularly effective against unprotected cryptographic IPs, micro-controllers and embedded systems [?].

Many software and hardware countermeasures have been developed since the early 1990s, but masking has received much attention in the last decade. Masking consists in splitting into *shares* the secret data and all subsequent intermediate computation results deriving from the secrets, in order to break the statistical correlation between secrets and side-channel observations. Given an appropriate masking scheme and an appropriate leakage model of the target circuit, a masking countermeasure is sound [?], and can be targeted by formal approaches to verify implementations [?], [?]. Masking is applied to hardware and software implementations of cryptographic primitives. We focus on the compilation of software implementations in the rest of this paper.

The compiler is a central piece in a production workflow of software applications and systems. Standard compilers target only the functional properties of the compiled program, and as such they are likely to alter or break countermeasures [?]. In practice, compiled secured implementations would not offer the expected level of security [?]. Hence, to the best of our knowledge, countermeasures are still manually applied in high-security products, involving modifications and analysis of implementations at the assembly level. For these reasons, several works have proposed to automatically apply countermeasures within the compiler [?], [?], [?]. These works paved the way for an automatic application of the countermeasure, although they suffer from various limitations: they are either target-dependent, or they need to peel or unroll loops, or they require the use of a domain specific language. Moreover, cryptographic implementations targeted by masking often make use of constant tables indexed with values depending on some secrets, e.g. S-Boxes, referred to as lookup tables in the following. Automated masking of lookup table accesses within a compiler has not received much attention, and all existing compilation approaches handle them by creating masked tables, whose masks are refreshed regularly. However, such approaches are not sound in some cases [?].

In this paper, we propose an approach to apply first-order boolean masking within a general-purpose compiler that addresses the limitations aforementioned. Our contributions are the following:

1) we show how to handle control flow, which makes it possible to apply the countermeasure on code with loops without unrolling them or peeling them,
2) we show how to integrate in a compiler the masking of lookup tables with a polynomial interpolation approach based on [?],
3) we propose several optimisations to speed up the evaluation of interpolating polynomials,
4) we exploit a state-of-the-art formal verification tool to check the absence of variables statistically dependent on a secret in the compiled binaries.

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue.

This work was partially funded by the French National Research Agency (ANR) as part of the project PROSECCO funded by the program AAP-2015 under grant agreement ANR-15-CE39. It was also partially funded as part of the SARMENTI project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825325. This work was partially supported by the French national program "Programme Investissements d'Avenir IRT Nanoelec" ANR-10-AIRT-05.

N. Belleville (corresponding author) and D. Couroussé are with the Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France (email: firstname.lastname@cea.fr)

K. Heydemann, Q. Meunier and I. Ben El Ouama are with the Sorbonne Université, CNRS, LIP6, F-75005, Paris, France (email: firstname.lastname@lip6.fr)

Experimental results show that our compiler correctly applies the countermeasure: the resulting binaries are functional, and the formal verification concludes that all the variables are statistically independent of secret. Furthermore, the optimisations targeting the interpolating polynomial evaluation result in a $\times 2$ speedup compared to the original approach. Compared to an access to a masked table followed by a mask refresh, the polynomial evaluation is up to $\times 2.4$ as fast. The C files used for experimental evaluation, as well as bytecode and object files produced, the polynomials generated and the logs of the formal verification analysis are available online[1].

This paper is structured as follows. Section II first gives some background. Section III presents the related works. Section IV describes our approach to apply the countermeasure on an intermediate code representation, including our algorithms to handle loops and our optimisations for fast polynomial evaluation. Section V is dedicated to the implementation of the approach in LLVM. Then, Section VI presents the results of a leakage analysis performed with a formal verification tool to check that binaries produced with our compiler verified the property that all variables are statistically independent of secrets. Section VII details our experimental results. In particular, Section VII-B is dedicated to performance of approaches to mask tables and to the study of our optimisations, and Section VII-C takes a wider look at performance of masked code depending on the nature of the instructions to be masked. Section VIII compares our approach with the related works. Finally, Section IX concludes.

II. Background

A. Masking countermeasure

Masking is a protection principle designed to remove the statistical correlation between side-channel observations and the secret data manipulated by a circuit or a processor [**?**], [**?**]. The principle is to modify the algorithm of the targeted program so that all intermediate results that depend on the secret data are separated into several parts (called *shares*), chosen such that the results can be recomposed using all the shares, and such that any subset of shares of size strictly less than the total number of shares is statistically independent of the secrets. A masking is said to be of order d if it splits secrets into $d+1$ shares.

The secret splitting is done by choosing d random numbers (the masks) $\{m_1, \ldots, m_d\}$ and by computing $s_{d+1} = v \odot m_1 \odot \cdots \odot m_d$ where $\odot$ is the operation used for masking, v is the secret variable to be masked, which gives $d+1$ shares (d masks and s_{d+1}). We focus in this paper on boolean masking, where $\odot$ corresponds to the exclusive OR $\oplus$ (also noted XOR).

With first-order boolean masking, a secret s is transformed into 2 shares s_0 and s_1 with $s_0 = s \oplus m$ and $s_1 = m$ where m is a random number. The masks are randomly chosen at each execution, so that the values of the shares change in a random way from one execution to another. All the subsequent computations manipulate separately the 2 shares, and the final value is reconstructed from its shares at the end. The transformation of the computations that manipulate shares is presented in Section II-C.

If 2 variables masked with the same masks are combined, a secret dependent leakage can appear. To solve this problem, a mask refresh is done on one of the variables: a new mask is generated and combined with the shares of the variable.

B. Threat model

The physical implementation of an algorithm can be modelled as a sequence of physical elementary computations [**?**]. In a physical implementation $\mathcal{I} = \{(C_i, f_i)\}$ of an algorithm, each physical elementary computation (C_i, f_i) is composed of an elementary computation C_i and of a leakage function f_i. A cryptographic implementation computes outputs from some inputs X and a secret k. Each execution of $\mathcal{I}$ leaks all the values $f_i(x)$, referred to as $(f_i(x))_i$ in the following, where x denotes X, k or any intermediate computation value deriving from X and k.

In the side-channel setting, the attacker can observe or control a set of inputs $\{X_j\}$. For each X_j, she can measure some physical quantities deriving from $(f_i(x_j))_i$, which we assume equivalent to $(f_i(x_j))_i$ for the sake of simplicity. A side-channel analysis tries to establish a statistical correlation between x_j and $(f_i(x_j))_i$, which leads to the value of the secret k. Note that it is also possible to exploit the knowledge of the computation outputs and the leakages f_i.

In an implementation $\mathcal{I}' = \{(C'_i, f'_i)\}$ protected with masking, each execution of $\mathcal{I}'$ leaks the values $(f'_i(x_j, R_j))_i$ where R_j is a set of unpredictable values (usually random) unknown to the attacker. The combination of the computation variables with random data breaks the statistical correlation between f'_i and the secret k, thus thwarting the attack. Higher-order attacks combine several observations of f'_i to recover (x_j, R_j), hence k. Their computational cost is however exponential with the order of the attack (i.e., the number of observations combined). To succeed, the attack order needs to be greater than the masking order d of the countermeasure.

In this paper, we consider an attacker capable of first-order attacks, and a value-based leakage model at the ISA level: leakages f_i depend on the independent values manipulated in the processor elements visible at the abstraction level of the Instruction Set Architecture (ISA), i.e., processor registers and memory.

C. Masking linear and non-linear operations

Each operation of the original program needs to be transformed so that it operates separately on each share, which depends on the nature of the operation: we distinguish linear and non-linear operations.

An operation f is linear w.r.t. boolean masking if: $\forall a, b : f(a \oplus b) = f(a) \oplus f(b)$. The transformations of operations that are linear w.r.t. boolean masking is done by applying the operation on each input share. For instance with 2 shares, a XOR between 2 secrets $s = a \oplus b$ becomes $s_0 = a_0 \oplus b_0$ and $s_1 = a_1 \oplus b_1$.

Non-linear operations, on the contrary, do not follow the linearity condition. For example:

[1]https://github.com/nicolas-belleville/Maskara

- XOR between a secret and a public value,
- boolean AND between 2 secret values,
- loads to a constant table, indexed by a secret variable.

To the best of our knowledge, there is no general approach for the transformation of non-linear operations, and, as a consequence, their masking needs to be examined on case-by-case basis. There has been a large amount of research to try to find efficient transformations for them [?], [?], [?], [?], [?], [?], [?], [?], [?]. Among them, the so-called *SecMult* algorithm [?] is used to mask the finite field multiplication between secrets, at the price of 4 finite field multiplications, 4 XOR, and one call to the random number generator. This masking scheme is adapted from the masking scheme of the boolean AND presented in [?]. In this section, we emphasise on one specific transformation required for boolean masking: loads to constant tables, which can be masked using several approaches. Other transformations will be detailed in Section IV-D.

D. Masking table lookups indexed by secrets

1) Overview: Two main approaches are usually used to mask loads to constant tables that are indexed by a secret [?], [?], [?], [?], [?], [?], [?]: (1) doing masked loads to a masked table [?], [?], [?]; (2) replacing the table by the masked evaluation of an interpolating polynomial [?], [?], [?], [?].

At first-order, the first solution consists in choosing 2 masks: one to mask the inputs of the table, and one to mask the outputs of the table. The table is in this case recomputed knowing these masks. However, mask refreshes force to recompute the table regularly, and an attacker may exploit the table recomputation to find the masks used [?].

The second solution consists in finding an interpolating polynomial of the table, and to mask the evaluation of this polynomial. The polynomial is defined in a finite field where addition is a XOR. This solution is more complex to set up than the first solution, but avoids the problem of table recomputation for mask refreshes.

Other *ad hoc* approaches target specific tables, like the approach proposed by [?] for the AES S-box, but we target here a generic solution able to mask any lookup table within the compiler.

2) Using polynomial interpolation: We introduce in this section the polynomial interpolation strategy to mask accesses to lookup tables.

We choose the finite field as the smallest finite field that contains all input and output elements of the lookup table, with the XOR as the addition, and a finite field multiplication defined using an irreducible polynomial. The interpolating polynomial will be defined using this finite field.

A generic approach would be to find an interpolating polynomial under the following form, for instance using a Lagrange polynomial (Eq. 1). However, the evaluation of P in Eq. 1 is expensive since it requires to compute all monomials, resulting in a lot of non-linear multiplications.

$$P[X] \quad = \quad \sum_{i=0}^{n-1} a_i X^i \tag{1}$$

Coron *et al.* [?] proposed instead to determine an interpolating polynomial in the form shown in Eq. 2. In this formula, t is a variable of the approach, which is chosen to have a few masked multiplications (using SecMult), as the latter are costly. The $t-1$ polynomials q_i and t polynomials p_i are polynomials constructed using mainly the squaring operation, as squaring a secret variable is a linear operation, whereas performing a finite field multiplication between 2 secret variables is not linear and requires a SecMult. This approach resulted in competitive performance as compared to the use of masked tables for first-order and higher-order masking [?]. We choose to implement this approach to handle table lookups indexed by secrets within our masking compiler pass.

$$P[X] \quad = \quad \sum_{i=1}^{t-1} p_i(X) \cdot q_i(X) + p_t(X) \tag{2}$$

III. Related works

Several works have considered the use of compilation to apply countermeasures against side-channel attacks [?]. Among them, [?], [?], [?] proposed different approaches to apply boolean masking within a compiler.

Moss *et al.* [?] proposed a first approach based on a type system and a static analysis to automatically mask a code written in a domain specific language. They unroll the loops and inline functions before the countermeasure is applied.

Agosta *et al.* [?] rely on a data-flow analysis that keeps track, for each variable, of the number of bits of the key that the variable depends on. This enables to mask only variables that could be exploited by an attacker, without masking the variables that depend on enough bits of the key to be out of computational reach. They use loop peeling when the analysis does not manage to propagate information, which means that they iteratively unroll 1 iteration of the loop at a time until the analysis succeeds.

The approach proposed by Bayrak *et al.* [?] differentiates in several ways: it can use either a static analysis to determine which instructions to mask, or a dynamic analysis by making the link between a measurement with the assembly code to determine which instructions induce leakage. The countermeasure is then applied only on instructions that leak information about the secret. Their compiler use is not standard, as they use a compiler to decompile assembly code, reconstructing a higher level representation where the countermeasure is applied before emitting binary code.

All of these approaches use a masked table approach to mask the loads indexed by a secret, but as said in previous section, this approach requires to recompute the table regularly which can be exploited by an attacker [?].

In this paper, we propose to tackle the problem of automated application of the countermeasure without control flow modification, as well as the problem of efficiently masking the loads indexed by a secret. For this second point, we show how to avoid the use of masked tables using a polynomial interpolation approach for which we present several optimisations. Our approach works with C code, and is target independent.

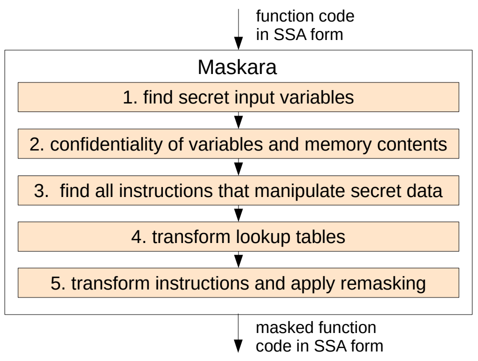
function code
in SSA form
Maskara
1. find secret input variables
2. confidentiality of variables and memory contents
3. find all instructions that manipulate secret data
4. transform lookup tables
5. transform instructions and apply remasking
masked function
code in SSA form

Fig. 1. Main steps for the application of the countermeasure

IV. Automated application of the first-order boolean masking countermeasure

A. Overview

In order to automatically apply the first-order boolean masking countermeasure during compilation, we developed several algorithms operating on a low-level Intermediate Representation (IR). We assume that the IR is target-independent and in Static Single Assignment (SSA) form, as implemented in production compilers like GCC and LLVM. In SSA form, each definition target must be a unique variable name. As a variable may be defined differently according to the execution path taken by the program, to respect the unique naming convention, so-called ϕ nodes are instructions inserted at control flow merge points. ϕ nodes select the incoming value according to the taken path and define a fresh variable set to this value. E.g., if a basic block BB2 has two predecessors BB0 and BB1 defining a source variable x, the basic block BB2 will first contain $\mathsf{x}_2 = \phi((\mathsf{x}_0, \mathsf{BB0}), (\mathsf{x}_1, \mathsf{BB1}))$, selecting the value of x in the SSA variable x_2 depending on the taken path.

The countermeasure is applied to each function independently, as illustrated in Figure 1. It is composed of several steps:

1) The identification of source variables specified as secret by the user.
2) A confidentiality analysis propagating this information to determine a level of confidentiality for memory contents at each program point and for all SSA variables.
3) An analysis that identifies the list of instructions LInst which manipulate secret data. The analysis is based on the results of the confidentiality analysis from the previous step, and is performed in a topological order defined w.r.t. the function instructions dependencies. Those instructions will be transformed to work with shares. This step also detects if LInst contains any load to constant lookup tables indexed by secrets as a preliminary to the next step.
4) The creation of specific functions, in the case where LInst contains accesses to constant lookup tables indexed by secrets. A new function is created for each lookup table. It performs a masked evaluation of a polynomial that interpolates the original table.
5) Transformation of each instruction of LInst to make it compliant with the masking scheme used (first-order boolean in the context of this paper). Tightly coupled with the transformation, a remasking analysis determines if the transformation may create a secret data leak. If this is the case, a remasking is performed.

The following sub-sections detail the steps 2, 4 and 5.

B. Confidentiality analysis

The first step of the masking pass starts by retrieving source variables specified as secret by the user (Fig. 1 step 1). This information is used to analyse the confidentiality of 1) the memory contents accessed by the program at each program point and 2) all SSA variables defined within the function using a data-flow analysis presented in this section (step 2). This confidentiality analysis is mandatory to determine the list of instructions that manipulate secret data (step 3).

We assume a reduced SSA IR language given in Figure 2 to explain the confidentiality analysis. The IR only comprises instructions that define a (new) SSA variable or change the memory contents, as other kinds of IR instructions do not impact the confidentiality of variables (e.g. jump instructions). We omit calls to functions returning a value for the sake of simplicity without loss of generality. Thus, our SSA IR language is composed of computational instructions (e.g., addition, boolean AND, shift operation, type conversion), ϕ node instructions selecting between SSA variables and memory access instructions performing a read or a write given an expression of the memory address to access. Expressions are either a SSA variable, a constant, a memory address corresponding to a source variable, or a computation on expressions.

$$\begin{array}{rl}
\mathsf{expr} := & \mathsf{var}_{\mathsf{SSA}} \mid \mathsf{cst} \mid @\mathsf{var}_{\mathsf{src}} \mid \\
& \mathsf{op}_{\mathsf{unary}}\ \mathsf{expr} \mid \mathsf{expr}\ \mathsf{op}_{\mathsf{binary}}\ \mathsf{expr} \\
\\
\mathsf{inst} := & \mathsf{store}(\mathsf{expr}, \mathsf{expr}_{\mathsf{addr}}) \\
& \mathsf{var}_{\mathsf{SSA}} = \mathsf{load}(\mathsf{expr}_{\mathsf{addr}}) \\
& \mathsf{var}_{\mathsf{SSA}} = \mathsf{oper}(\mathsf{expr}_1, ..., \mathsf{expr}_n) \\
& \mathsf{var}_{\mathsf{SSA}} = \phi(\mathsf{var}_{\mathsf{SSA}_1}, ..., \mathsf{var}_{\mathsf{SSA}_n})
\end{array}$$

Fig. 2. Expression and IR languages

Let $\mathcal{T}$ be a set of confidentiality types composed of $\{\mathcal{S}_i | i \geq 0\} \cup \{\mathcal{P}\}$. Type $\mathcal{S}_0$ corresponds to a secret data type, $\mathcal{S}_i,\ i > 0$ represents the type of memory addresses containing (or of pointers to) data of type $\mathcal{S}_{i-1}$. Type $\mathcal{P}$ corresponds to public data. For keeping uniform all the rules for the confidentiality analysis we pose $\mathcal{P} \equiv \mathcal{S}_\infty$ and get $\mathcal{T} = \{\mathcal{S}_i | i \geq 0\} \cup \{\mathcal{S}_\infty\}$. In the sequel, we use the terms *confidentiality type* and *confidentiality* interchangeably, and we say that the confidentiality gets higher if i decreases (the variable becomes closer to a secret), and that the confidentiality gets lower if i increases.

We assume that, at the beginning of a function, alive source code variables, parameters, alive memory addresses or alive memory contents have a type in $\mathcal{T}$. It is either defined by the programmer or derived from the confidential analysis of the calling function. By default, variables and addresses are considered public, hence of type $\mathcal{S}_\infty$, unless the user declares another type (this can be done using simple annotation), or the inference algorithm determines a higher confidentiality.

$$\frac{\mathsf{isConst}(\mathsf{e})}{\Gamma \vdash \mathsf{e} : \mathcal{S}_\infty}\ (\textsc{cst}) \qquad \frac{\mathsf{isSrcVar}(\mathsf{var}_{\mathsf{ssa}}) \quad \Gamma \vdash \mathsf{SrcVar}(\mathsf{var}_{\mathsf{ssa}}) : \mathsf{t}}{\Gamma \vdash \mathsf{var}_{\mathsf{ssa}} : \mathsf{t}}\ (\textsc{src}) \qquad \frac{\Gamma \vdash \mathsf{var}_{\mathsf{src}} : \mathcal{S}_i}{\Gamma \vdash @\mathsf{var}_{\mathsf{src}} : \mathcal{S}_{i+1}}\ (\textsc{srcaddr})$$

$$\frac{\mathsf{Mem}[\mathsf{addr}] = \mathsf{e} \quad \Gamma \vdash \mathsf{e} : \mathcal{S}_i}{\Gamma \vdash \mathsf{addr} : \mathcal{S}_{i+1}}\ (\textsc{addr}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathcal{S}_i}{\Gamma \vdash \mathsf{op}_{\mathsf{unary}}\ \mathsf{e} : \mathcal{S}_i}\ (\textsc{op1}) \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \mathcal{S}_{k_1} \quad \Gamma \vdash \mathsf{e}_2 : \mathcal{S}_{k_2}}{\Gamma \vdash \mathsf{e}_1\ \mathsf{op}_{\mathsf{binary}}\ \mathsf{e}_2 : \mathcal{S}_{min(k_1,k_2)}}\ (\textsc{op2})$$

Fig. 3. Confidentiality typing rules for expressions

$$\frac{\forall i \in [1,n]\ \Gamma \vdash \mathsf{e}_i : \mathcal{S}_{k_i}}{\Gamma, \mathsf{v} := \mathsf{oper}(\mathsf{e}_1, ..., \mathsf{e}_n) \to \Gamma[\mathsf{v} \leftarrow \mathcal{S}_{\min\limits_{1 \leq i \leq n} k_i}]}\ (\textsc{comput}) \qquad \frac{\forall i \in [1,n]\ \Gamma \vdash \mathsf{v}_i : \mathcal{S}_{k_i}}{\Gamma, \mathsf{v} := \phi(\mathsf{v}_1, ..., \mathsf{v}_n) \to \Gamma[\mathsf{v} \leftarrow \mathcal{S}_{\min\limits_{1 \leq i \leq n} k_i}]}\ (\textsc{phinode})$$

$$\frac{\Gamma \vdash \mathsf{e}_{\mathsf{addr}} : \mathcal{S}_i}{\Gamma, \mathsf{v} := \mathsf{load}(\mathsf{e}_{\mathsf{addr}}) \to \Gamma[\mathsf{v} \leftarrow \mathcal{S}_{\max(0,i-1)}]}\ (\textsc{load}) \qquad \frac{\Gamma \vdash \mathsf{e} : \mathcal{S}_i}{\Gamma, \mathsf{store}(\mathsf{e}, \mathsf{e}_{\mathsf{addr}}) \to \Gamma[\mathsf{e}_{\mathsf{addr}} \leftarrow \mathcal{S}_{i+1}]}\ (\textsc{store})$$

Fig. 4. Confidentiality propagation rules for instructions

Let Γ be the environment associating memory contents, source variables, or defined SSA variables to a confidentiality type $\mathcal{T}$. Figure 3 defines the confidentiality inference rules for expressions of our IR language. We note $\Gamma \vdash \mathsf{e} : \mathsf{t}$ the confidentiality t of expression e according to Γ. The predicate $\mathsf{isSrcVar}$ tests if a SSA variable corresponds to a source variable (be it a parameter, a local or a global variable), and the function SrcVar returns this variable in affirmative case. The rules in Figure 4 reflect how instructions of our IR language update the environment. Notation $\Gamma, \mathsf{I} \to \Gamma'$ means that the update of Γ by executing I gives the environment Γ'. We note $\Gamma[\mathsf{v} \leftarrow \mathsf{t}]$ the update of the confidentiality of v with the confidentiality type t.

Using the defined rules, the confidentiality analysis follows a forward data-flow analysis to infer the type of each SSA variable defined by a function: from the entry of a function, each instruction is processed by following the control-flow graph. Due to the presence of loops, the analysis is iterative until every SSA variable is typed and a fixed point is reached. Note that the confidentiality of a SSA variable defined by an instruction cannot be lower than the confidentiality type of the instruction operands (rules COMPUT, PHINODE, LOAD). The analysis can never derive a public data from a secret.

C. *Polynomial interpolation of lookup tables*

This section shows how constant tables that are accessed with indexes depending on secret data are transformed (Fig. 1 step 4). Lookup tables include, in particular, substitution tables, called *S-boxes* in cryptographic primitives whose structure is a substitution/permutation network that associates to each integer of a set another integer of another set. For example, in the case of AES, the S-box can be seen as a bijection of the set of integers in the $[0, 255]$ interval in itself.

1) Overview: The use of lookup tables is detected by the presence of a load whose base address corresponds to a constant table and whose index is a secret (type $\mathcal{S}_0$). We chose to mask lookup tables by creating for each of them a function that performs a masked evaluation of an interpolating polynomial defined over a finite field. We follow the method of Coron *et al.* [**?**] to construct an interpolating polynomial in the form presented in Section II-D. This construction is detailed in Section IV-C2. The generation of the code that handles the masked evaluation of the polynomial is presented in Section IV-C3. Finally, in Section IV-C4, we bring several optimisations to speed up the evaluation of the polynomial.

2) Polynomial construction: As proposed in [**?**], finding an interpolating polynomial as a sum of products of some polynomials p_i and q_i (Eq. 2) makes it possible to avoid computing all monomials and to reduce the number of non-linear multiplications when evaluating it. The q_i and p_i are constructed using only a chosen restricted set of monomials denoted $\mathcal{M}$, which must satisfy condition $Cond_{\mathcal{M}}$ as follows: for all monomial m not in $\mathcal{M}$, there exists a product of 2 monomials p, q in $\mathcal{M}$ such that $m = p \cdot q$.

The set $\mathcal{M}$ is the union of two sets $\mathcal{M}_{NL}$ and $\mathcal{M}_S$ that are constructed as follows. $\mathcal{M}_{NL}$ is first built to contain a fixed number l of monomials: it comprises X^0, X^1 and $l-2$ other monomials that can only be obtained using a non-linear multiplication. $\mathcal{M}_S$ contains all the monomials that can be obtained from $\mathcal{M}_{NL}$ only using squaring operations. The chosen monomials in $\mathcal{M}_{NL}$ and the derived ones in $\mathcal{M}_S$ are kept only if the resulting set $\mathcal{M}$ satisfies $Cond_{\mathcal{M}}$. Otherwise, a different set of monomials $\mathcal{M}_{NL}$ must be chosen and so until $\mathcal{M}$ satisfies $Cond_{\mathcal{M}}$.

We follow [**?**] in the choice of the number l of monomials as well as the number t of p_i, which depend on the size of the table to interpolate. Thus, as the choice of the set of monomials $\mathcal{M}_{NL}$ is independent of the table content, the compiler could use pre-computed static sets of monomials for various table sizes instead of computing a valid set at each new compilation.

The coefficients of each q_i are then chosen randomly and the equation $\forall x : P[x] = table[x]$ gives a set of linear equations, with p_i's coefficients as unknowns. The system is solved to find the p_i's coefficients. There is a very high probability that the system is full rank and has a solution due to the random nature of the coefficients of each q_i. In case the system happens to have no solution, the algorithm chooses different coefficients for the q_i and starts a new search.

Once all the coefficients have been determined, the code performing the evaluation of the interpolating polynomial can be emitted.

3) Code generation: The approach presented above gives an analytical expression of the polynomial. From this expression, we construct a function that performs the masked evaluation of the polynomial. A general purpose compiler is highly effective at optimising the program structure and computations, but domain-specific optimisations are much harder to implement in such a compiler. Hence, instead of emitting the code performing an evaluation of the polynomial and mask it in 2 separated steps, we choose to emit directly a code performing the masked evaluation of the polynomial. Doing so enables us to apply algorithmic optimisations based on the mathematical properties of the operations used in the polynomial. More specifically, we know exactly the location of the finite field multiplications, and we can then handle them directly using the SecMult algorithm.

The code generation starts by emitting masked code for each monomial in $\mathcal{M}_{NL}$ using SecMult for each non-linear multiplication. Then, the masked code of each q_i and p_i is emitted, using squaring, multiplications by constants and XORs. Finally, the SecMults and XORs necessary to multiply the values of the q_i and p_i and to sum the results are emitted.

4) Optimisations of polynomial evaluation: In order to make the polynomial evaluation faster, we reuse some existing optimisations and bring new ones:

- some linear operations are grouped together and tabulated,
- some of the coefficients of the q_i polynomials are set to zero to make the evaluation of q_i faster,
- the number of operations for finite field multiplication using log/alog tables is reduced by expanding these tables.

a) Grouping of operations: The first optimisation, proposed in [**?**], consists in grouping linear operations to form larger linear expressions that are tabulated. Within each polynomial p_i and q_i, all monomials that can be derived from a same monomial from $\mathcal{M}_{NL}$ through squaring are grouped together. We will call such group of monomials a *class* later on. Then, p_i and q_i are written as a sum of subpolynomials $\sum_j P_{S_j}$. For each subpolynomial $P_{S_j}[X] = a_k X^\alpha + a_{k+1}(X^\alpha)^2 + a_{k+2}(X^\alpha)^4 + \ldots$, we define the subpolynomial $P'_{S_j}[X] = a_k X + a_{k+1}X^2 + a_{k+2}X^4 + \ldots$. This enables to compute $P_{S_j}[X]$ in two steps:

1) By computing $Y = X^\alpha$;
2) By evaluating the polynomial $P'_{S_j}[Y]$

X^α is not linear w.r.t. XOR, but P'_{S_j} is linear because it is composed only of powers of 2, scalar multiplications, and XORs. Consequently, once $Y = X^\alpha$ has been evaluated and its shares y_0 and y_1 are known, one obtains the shares of the result by evaluating P'_{S_j} on both shares. Even though P'_{S_j} is only constituted of linear operations that are easy to mask, the code is made faster by tabulating the result of P'_{S_j}. This enables to get $P'_{S_j}[y_0]$ and $P'_{S_j}[y_1]$ using 2 table lookups once the shares y_0 and y_1 are known.

b) Choice of coefficients of q_i: In the original approach, coefficients of q_i are random numbers, which allows the linear equation system to be full rank with high probability. We propose an optimisation in the choice of the coefficient of q_i: it consists in zeroing a large part of them to avoid some computations. As all the operations related to a class are grouped to form subpolynomials P_{S_j}, our algorithm either sets to zero all the coefficients of a class, or sets them to random values.

We propose a heuristic to choose the coefficients of q_i to be zeroed. It is described in Algorithm 1. The user gives a target sparsity `sp` for the algorithm, which corresponds to the mean number of classes the user wants to be zeroed per q_i. Sparsity `sp` is not necessarily an integer, but it must be lower to $l - 1$ in order to never zero all coefficients of a q_i.

For each q_i (variable `qi` in Algorithm 1), the algorithm first chooses randomly its coefficients (setRandomly(`qi`)). Then it randomly chooses a number `nC` in the interval [0,`l`$-$1[, in a way that gives a theoretical mean of `sp`. This number `nC` correspond to the number of classes randomly selected among all classes and whose coefficients are zeroed by setNClassesToZero(`qi`, `nC`).

Once the coefficients of `qi` have been chosen, the compiler tries to solve the system of linear equations. The setting of coefficients to zero can lead to unsolvable systems. In such case, our compiler tries again setting to zero a different set of coefficients, until it finds a solvable system.

The heuristic finds as many different sets of coefficients that the user asks for (`nbIt` parameter). The heuristic then selects the set of coefficients that will give the fastest resulting implementation.

This heuristic has a probabilistic termination that depends on `sp`. The closer to `l`$-$1 the value of `sp` is, the sparser the set of coefficients is; and as a consequence, the faster the resulting implementation. However, the compiler may need a lot of attempts before finding a suitable set of coefficients. In practice, the user can adjust `sp` and the number of iterations depending on its needs. In addition, the algorithm could keep the best seed (called `it` in Algorithm 1) of the PRNG to find directly this efficient solution without iterating.

c) Optimisation of finite field multiplications with log/alog tables: Finite field multiplication is used extensively when computing the masked evaluation of the interpolating polynomial: once for each linear multiplication and four times for each non-linear multiplication carried out using SecMult. Hence, finite field multiplication becomes an interesting target for performance optimisations. We present here several optimisations to accelerate its computation.

We chose to implement finite field multiplication using log/alog tables. This approach relies on the property that any element of the finite field, except 0, can be represented as a power of the primitive element e of the finite field:

$$\forall x \neq 0 : \exists p : x = e^p \tag{3}$$

Multiplying two elements x_0 and x_1 in a binary finite field of 2^n elements then consists in the following steps:

1) if any of the operands equals 0, return 0
2) for each x_i, find exp_i such that $x_i = e^{exp_i}$
3) compute $exp_{res} = exp_0 + exp_1[2^n - 1]$

Algorithm 1: Heuristic for the optimisation of qi coefficients

```
{qi, pi} ← optimisedPolynomialInterpolation(S, sp, nbIt)
    Input: Lookup table to interpolate S
    Input: Target sparsity sp
    Input: Number of iterations nbIt
    Output: Set of pi and qi that interpolate S
1   t = getNumberOfpi(S)
2   l = getNumberOfBasisExponents(S)
3   bestqipi = {}
4   lowestCost = +∞
5   for it = 0; it < nbIt; it += 1 do
6       srand(it) // initialize PRNG
7       piSet = ∅
8       do
9           qiSet = ∅
10          for i = 0; i < t - 1; i += 1 do
11              setRandomly(qi)
12              // choose an int randomly
                 in [0,l−1[ with an average of sp
13              nC = randInt(0, sp, l - 1)
14              //
                 set to 0 nC classes in qi, other
                 coefficients are left random
15              setNClassesToZero(qi, nC)
16              qiSet.insert(qi)
17          // try to solve system;
             return an empty set if no solution
18          piSet = solveSystem(qiSet, S)
19      while piSet == ∅
20
21      cost = estimateCost(piSet, qiSet)
22      if cost < lowestCost then
23          lowestCost = cost
24          bestqipi = {qiSet, piSet}
25  return bestqipi
```

4) compute x_{res} such as $x_{res} = e^{exp_{res}}$

The step 1 is necessary as 0 does not have any logarithm. The steps 2 and 4 are implemented with two lookup tables: the `log` table associating to a non-zero element in the finite field its logarithm, and the `alog` table associating to a power p the value of e^p. We use the `log` and `alog` notations in the following for these tables to differentiate them from the log and $alog$ mathematical functions.

To avoid leaking the operand values through execution time, the implementation has to be constant time: the same computation must be performed whether the operand is null or not. Thus, a classical constant-time implementation of finite field multiplication in C would be:

```
uint8_t mul(uint8_t a, uint8_t b) {
  return ((a != 0) & (b != 0))
  * alog[(log[a] + log[b]) % ((1<<n)-1)];
}
```

As previously proposed by [**?**], the modulo computation can be optimised by doubling the size of the `alog` table and by letting: $\forall x \geq (2^n - 1) : \texttt{alog}[x] = \texttt{alog}[x - (2^n - 1)]$.

We propose a further optimisation in order to entirely remove all the code that handles the special case where one operand is null. We notice that $\forall x \neq 0 : \ \texttt{log}[x] \leq (2^n - 1)$. Thus, $\forall x \neq 0, \forall y \neq 0 : \ \texttt{log}[x] + \texttt{log}[y] \leq 2 \cdot (2^n - 1)$. As a consequence, by modifying the `log` table so that $\texttt{log}[0] = 2^{n+1} - 1$, we get: $\forall x \ : \ 2 \cdot (2^n - 1) < \texttt{log}[0] + \texttt{log}[x] \leq 2 \cdot (2^{n+1} - 1)$. By doubling the size of the `alog` table by setting $\forall x > 2 \cdot (2^n - 1) \ : \ \texttt{alog}[x] = 0$, we can remove the code handling the case where one operand is null. The implementation of the multiplication in C becomes:

```
uint8_t mul(uint8_t a, uint8_t b) {
  return alog[log[a] + log[b]];
}
```

Note that one can choose for $\texttt{log}[0]$ any value that is bigger than any sum of the 2 logs.

D. Transformation of operations

The transformation of instructions (Fig. 1 step 5) is carried out instruction per instruction. After each transformation, the pass checks if a leakage appeared and if so adds a remasking of the operands before the leaky instruction. This step is explained in section IV-E.

1) Transformation of linear operations: Linear operations are transformed by applying the same operation on both shares. Examples of linear operations are: load from and store to an address of type $\mathcal{S}_1$, XOR between 2 masked variables, sign extend and zero extend of a masked variable, shift of a masked variable, boolean AND between a masked variable and a public variable, ϕ node that has 2 secret operands.

2) Transformation of non-linear operations: Non-linear operations are transformed using operation-specific recipes. Examples of non-linear operations are: XOR between a masked variable and a public variable, boolean AND between 2 masked variables, boolean OR between 2 masked variables, boolean OR between a masked variable and a public variable, loads indexed by a secret.

The transformations are very diverse depending on the case:

1) XOR between a masked variable and a public variable does not need any transformation
2) the boolean AND between 2 masked variables is masked using the ISW scheme [**?**]
3) the boolean OR between 2 masked variables is masked using the SecOr formula from [**?**]
4) the boolean OR between a masked variable and a public variable is transformed using the following property: $a \vee (s_0 \oplus s_1) = (a \vee s_0) \oplus (\neg a \wedge s_1)$
5) loads to a constant table indexed by a secret are replaced by a function call to the function created for this table (see section IV-C).

E. Remasking analysis

The remasking analysis determines if an instruction leaks secret data. The analysis must operate after the transformation of operations presented above, because the transformations may introduce new instructions and new intermediate variables. A leakage appears if one of the result of the instructions is not correctly masked. When it happens, a remasking (also called *mask refresh*) of one operand of the leaky instruction is needed. Note that the remasking is simultaneously applied to one operand and its associated share in order to maintain the correctness of the share decomposition.

Our remasking analysis is a forward data-flow analysis (from input to output). It aims to conservatively compute, for each SSA variable produced by a transformed instruction, the potential lists of its masks. More precisely, this analysis attaches to any such SSA variable v a set of potential lists of masks for the variable as a whole denoted $\mathcal{L}(\mathsf{v})$ and a set of potential lists of mask bits for each of its bits denoted $\mathcal{L}_i(\mathsf{v})$ for bit i. Maintaining these lists at the word level and at the bit level is necessary to correctly handle and propagate the masks in presence of shift or rotations, bit selections or sign extensions.

We suppose that each function's secret input is already decomposed into two shares and that one of the share is a mask. This enables to initialise, at the beginning of the analysis, the set of the masks list of each input variable. The analysis then follows mask propagation rules to determine, for each instruction that manipulates one or several shares, the set of the potential lists of masks $\mathcal{L}(\mathsf{r})$ and $\mathcal{L}_i(\mathsf{r})$ of its result variable r.

In case of a boolean instruction with only one secret operand a, $\mathcal{L}(\mathsf{r})$ and $\mathcal{L}_i(\mathsf{r})$ are identical to $\mathcal{L}(\mathsf{a})$ and $\mathcal{L}_i(\mathsf{a})$ respectively. In case of shifts and sign extends, $\mathcal{L}(\mathsf{r})$ is set to $\mathcal{L}(\mathsf{a})$, and $\mathcal{L}_i(\mathsf{r})$ is a copy of the set of the corresponding bit of the input.

In case of a ϕ node $\mathsf{r} = \phi((\mathsf{a}, \mathsf{BB0}), (\mathsf{b}, \mathsf{BB1}))$. The set $\mathcal{L}(\mathsf{r})$ (and $\mathcal{L}_i(\mathsf{r})$ resp.) is set to the union of the set $\mathcal{L}(\mathsf{a})$ and $\mathcal{L}(\mathsf{b})$ ($\mathcal{L}_i(\mathsf{a})$ and $\mathcal{L}_i(\mathsf{b})$ resp.). The ϕ node justifies the computation of the set of the *potential* lists of masks for each variable. Depending on the execution path taken by the program, the result of the ϕ node will be masked either with the masks of a or with the ones of b.

For instructions that have two secret operands a and b, $\mathcal{L}(\mathsf{r})$ is set to the union of all the possible symmetric differences of a list lm_i in $\mathcal{L}(\mathsf{a})$ with a list lm_j in $\mathcal{L}(\mathsf{b})$:

$$\mathcal{L}(\mathsf{r}) = \bigcup_{\mathsf{lm}_i \in \mathcal{L}(\mathsf{a}), \mathsf{lm}_j \in \mathcal{L}(\mathsf{b})} \{\mathsf{lm}_i \triangle \mathsf{lm}_j\}$$

The symmetric difference of two mask lists denoted $\mathsf{lm}_i \triangle \mathsf{lm}_j$ equals to $(\mathsf{lm}_i \cup \mathsf{lm}_j) \setminus (\mathsf{lm}_i \cap \mathsf{lm}_j)$. The same computation enables to derive all the $\mathcal{L}_i(\mathsf{r})$. If any list in $\mathcal{L}(\mathsf{r})$ or any $\mathcal{L}_i(\mathsf{r})$ gets empty, one of the source operands must be remasked. The remasking will add a new mask (or a new bit mask resp.) in all the lists of $\mathcal{L}(\mathsf{r})$ ($\mathcal{L}_i(\mathsf{r})$ resp.). The set of masks lists $\mathcal{L}(\mathsf{r})$ and $\mathcal{L}_i(\mathsf{r})$ will then be recomputed.

Cyclic dependencies exist when a ϕ node depends on a variable that depends on the result of the ϕ node. Such ϕ nodes appear in the header of loops. When a ϕ node that has one operand which exhibits such a cyclic dependency, our analysis tries to determine a set of lists of masks for the result of the ϕ node using Algorithm 2. In this algorithm, we call the initial variable (`iv`) the operand of the ϕ node that does not have any cyclic dependency, and cyclic variable (`cv`) the operand that exhibits the cyclic dependency. The main idea of this algorithm is to simulate the mask propagation, as if the following instructions were transformed, but without transforming them in order to be able to compute the potential lists of masks for the cyclic variable `cv`. The algorithm iterates while the set $\mathcal{L}(\mathsf{r})$ of the result of the ϕ node (denoted `S` in Algorithm 2) grows. When the propagation does not add any new list of masks in this set, the algorithm has found the set of all possible lists of masks for the ϕ node (Alg. 2 line 7). However, the algorithm does not always converge. As such, the number of iterations must be bounded by the user (or with a default value). When the algorithm reaches this limit, it returns an empty set, and the results of the ϕ node will be remasked. This will update the ϕ node's mask at every iteration, ensuring the variable is correctly masked at the price of a potential higher performance cost.

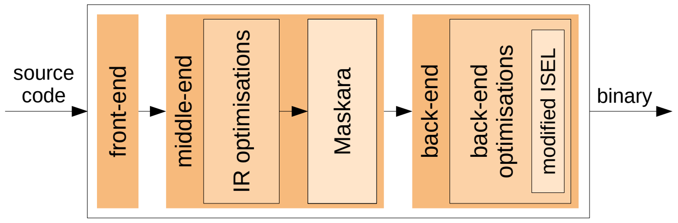

Fig. 5. Compilation flow to apply the countermeasure. The Maskara masking pass is inserted at the end of the middle-end. The back-end is slightly modified: the optimisation level of instruction selection (ISEL) is set to zero.

Algorithm 2: Construction of the set of lists of masks for a ϕ node that has cyclic dependencies

```
S ← getPhiNodeMaskSet(inst, timeout)
    Input: Instruction inst, integer timeout
    Output: Set of Mask Lists S
1   // get the operand which initialises the $\phi$ node value
2   iv = inst.getInitialVariable()
3   // get the operand which exhibits cyclic dependency
4   cv = inst.getCyclicVariable()
5   S.addListsOf(getSetOfMaskLists(iv))
6   for i = 0; i < timeout; i += 1 do
7       // determine the mask lists of all instructions until cv assuming S is the set of $\phi$ node's mask lists
8       propagateMaskListToInstructionsUntil(cv)
9       // test if mask lists from this iteration are already all in S
10      if S.includes(cv.getSetOfMaskLists()) then
11          return S
12      else
13          // update S
14          S.addListsOf(getSetOfMaskLists(cv))
15  return emptyList()
```

V. Implementation within LLVM

We implemented our approach in a compiler pass, called Maskara, within LLVM.

To use our pass, the user has to annotate its source code to declare the secret variables and the functions to protect. The pass then masks all the instructions that operate on secret data in the target functions. The pass aborts and raises an error if it finds instructions that manipulate secrets for which no transformation rule is implemented.

Maskara is placed at the very end of the middle-end, as presented in Figure 5. This insertion of the pass in the middle-end makes the application of masking independent of the target architecture, which greatly facilitates support of different platforms. Moreover, the countermeasure is applied after all the optimisation passes of the middle-end and can then benefit from them (smaller and more efficient code). Also, the application of the countermeasure before the back-end takes advantage of the register allocation which optimises the register use. This is particularly interesting for our countermeasure, since the application of masking splits many variables in shares, which increases the register pressure.

However, it is necessary to make sure that other back-end optimisations and passes do not degrade the masked code such that a leak appears. Thus, we investigated potential sources of threats in the back-end by considering the ARM back-end. Indeed, we chose to focus on the ARM back-end since all experiments were made on this architecture, including the formal verification. We found out that the selection of instructions from the ARM back-end can hurt the countermeasure when combining some instructions into a unique one for performance and code size reasons. As an example, the instruction selection can gather 2 one-byte memory loads at contiguous addresses to form a single two-byte (half-word) memory load. This is harmful when the 2 bytes to be loaded correspond to the two shares s_0 and s_1 of a secret value s. Indeed, the concatenation of 2 secrets that have the same mask can induce a leakage.

As an example, if we model the leakage using the Hamming weight of the value, and assume that $s_0 = s \oplus rand$ and $s_1 = rand$:

$$\begin{aligned} HW(s_0||s_1) &= HW(s_0) + HW(s_1) \\ &= HW(s \oplus rand) + HW(rand) \end{aligned}$$

where $||$ corresponds to concatenation here.

If a variable is correctly masked, any Hamming weight should be possible for any variable value. For $s = 255$, this property does not hold, as we get a constant Hamming weight:

$$\begin{aligned} HW(s_0||s_1) &= HW(\neg rand) + HW(rand) \\ &= 8 - HW(rand) + HW(rand) = 8 \end{aligned}$$

Thus, the Hamming weight of the half-word has a distribution that depends on the value of the secret: such a pattern introduces a vulnerability in the masked code.

We solved this issue by disabling optimisations that combine instructions in the instruction selection pass.

Apart from this optimisation, we did not detect other ARM back-end optimisations that could alter the countermeasure. We did not change the optimisation level for other back-end passes, letting the user defined levels. Moreover, in the next section, we show, using a formal verification approach, that the ARM binaries produced using optimisation level `-Oz` are well protected and thus confirm that the other back-end passes did not harm the countermeasure.

VI. Leakage analysis

The masking countermeasure at first-order aims at making all intermediate computations statistically independent of the secrets. If the countermeasure is correctly applied, there should not be any leakage of secret information in the value-based leakage model. We choose to formally check this property on representative binaries generated by our hardening compiler.

We carry out the analysis using a tool implementing the formal verification approach proposed in [**?**]. This verification approach analyses the statistical distribution of the values of masked expressions. It is based on a symbolic analysis and inference rules. The computed distribution can be either uniform (RUD), statistically independent of the secrets (ISD), statistically dependent on the secrets (SDD), constant (CST), or unknown (UKD). RUD, ISD and CST are leak-free. UKD means either that the expression is not perfectly masked, or that the verification was not able to determine the distribution of the variable. SDD highlights a vulnerability. The verification is performed on a per-function basis. The verification tool first recomputes the expression of each result of intermediate computation of the binary program. Then, it performs the symbolic analysis to determine the statistical distribution of the values of each intermediate computation of the binary code. The composition of two verified functions $f \circ g$ is sound if each secret output of the first function (g) has a uniform distribution (condition 1) and has at least one mask that does not appear in any other secret output (condition 2).

We consider all the functions of an AES implementation (Section VII-C): AddRoundKey, SubBytes, ShiftRows, MixColumns and KeySchedule. We also analyse the function InterpolatedSboxAccess, generated by the Maskara compiler pass, that performs one masked evaluation of a polynomial interpolating the AES S-box.

For all the functions, the formal verification tool concluded that all intermediate computation results are statistically independent of the secret. The analysis results are given in Table I: for all functions but MixColumn, all the intermediate computation results and all function outputs either have a uniform distribution (RUD) or are constant (CST). For the function MixColumn, all intermediate computation results have either a uniform distribution (RUD), a distribution independent of the secret (ISD) or are constant (CST). We checked that the outputs of MixColumns have a uniform distribution (RUD) (condition 1). Condition 2 is also systematically satisfied since the masks used for the remaskings are always fresh ones. We can then conclude that these functions can be safely composed and that the whole AES implementation is secure in the value-based leakage model. We can note that these security results are similar to those obtained with an approach using masked tables, as shown in [**?**].

We would like to point out that the first analysis that we carried out revealed some leakage in the InterpolatedSboxAccess function and so in the SubBytes and KeySchedule functions that call it. The verification enabled us to find a bug in the implementation of the compiler masking pass. The results that we present here have been obtained after we fixed our compiler pass implementation.

In the end, all the instructions manipulating secret information were correctly masked by the compiler. Moreover, the compiler back-end has not impaired the countermeasure. This result allows us to be confident that the countermeasure

TABLE I
VERIFICATION RESULTS: NUMBER OF ARM ASSEMBLY INSTRUCTIONS ANALYSED PER FUNCTION, AND BREAKDOWN OF THE COMPUTED DISTRIBUTIONS FOR INSTRUCTION RESULTS

		Distribution of instruction result				
Function	**#inst**	**CST**	**RUD**	**ISD**	**SDD**	**UKD**
AddRoundKey	181	53	128	0	0	0
InterpolatedSboxAccess	707	211	496	0	0	0
SubBytes	11332	3396	7936	0	0	0
ShiftRows	54	2	52	0	0	0
MixColumn	346	30	273	43	0	0
KeySchedule	2971	858	2113	0	0	0

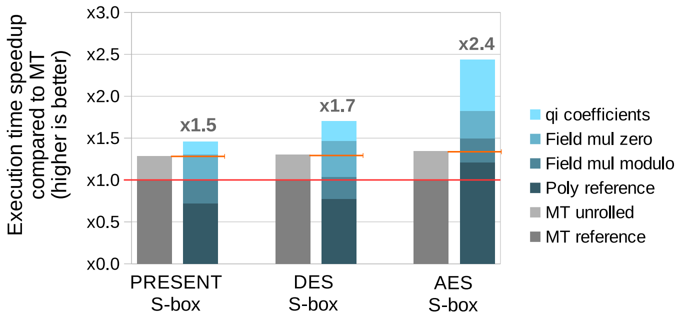

Fig. 6. Execution time speedup using our optimisations, compared to a masking table approach. Our approach is faster than the masking table approach on all test-cases.

is properly carried out by Maskara.

VII. EXPERIMENTAL EVALUATION

A. Experimental setup

We used a STM32VLDISCOVERY board from STMicroelectronics, fitted with a Cortex-M3 core, 8 kB of RAM, and 128 kB of flash memory. The execution time is measured in clock cycles using the hardware counters.

B. Performance evaluation of lookup table accesses

In this section, we evaluate the impact of the countermeasure on execution time for lookup table accesses, as well as the benefits of the optimisations proposed in Section IV-C4. As the number of q_i and of monomials involved in the interpolating polynomial depends on the finite field that contains the table elements, we chose tables that are built on different finite fields with various number of elements. We choose 3 different tables from widely used block ciphers as benchmarks: one 16-element S-box from PRESENT, one 64-element S-box from DES, and one 256-element S-box from AES.

For each table, we compare the execution times obtained using our optimisations and the polynomial interpolation approach with the execution times obtained using a masked table (`MT`) approach. To make a fair comparison between masked table and polynomial interpolation, we consider that masks are refreshed at every masked lookup table access, as it is the case for polynomial interpolation by default. For masked tables, this implies a table refresh after each access. The resulting speedups are given in Figure 6 and discussed below.

We measure the execution time of one masked table access and table refresh as a reference execution time denoted `MT reference`. The execution times obtained are 238 clock cycles, 711 clock cycles and 2637 clock cycles for PRESENT, DES and AES S-boxes respectively. We then enable loop unrolling that leads to a version denoted `MT unrolled`, which approximately exhibits a $\times 1.3$ speedup for all the test-cases.

TABLE II
EXECUTION TIME (PROC. CYCLES) OF THE EVALUATION OF SEVERAL S-BOXES ACCORDING TO THE OPTIMISATIONS APPLIED AND SPEEDUPS COMPARED TO THE POLYNOMIAL INTERPOLATION WITH ONLY THE GROUPING OF OPERATIONS

	Ref	Field mul modulo		Field mul modulo + zero		Field mul modulo + zero + qi coefficients	
S-box	time	time	speedup	time	speedup	time	speedup
PRESENT	331	239	$\times 1.4$	182	$\times 1.8$	163	$\times 2.0$
DES	920	685	$\times 1.3$	484	$\times 1.9$	417	$\times 2.2$
AES	2185	1761	$\times 1.2$	1448	$\times 1.5$	1083	$\times 2.0$

For the polynomial interpolation approach, we add progressively more optimisations. We start with an implementation using only the grouping of operations optimisation, called `Poly reference` in Figure 6. This implementation was slower than both `MT` implementations for PRESENT and DES S-boxes. However, it was already faster than `MT reference` for the AES S-box. Adding the optimisation of the modulo operation within the field multiplication gives the implementations denoted `Field mul modulo`, which are as fast or faster than the `MT reference` implementation for all test-cases. Yet, the `MT unrolled` implementation remains faster for both PRESENT and DES. Optimising again the field multiplication to remove the code handling the zero value makes the code, denoted `Field mul zero` in Figure 6, as fast or faster than `MT unrolled` for all the test-cases. Finally, the implementations obtained by adding the optimisation of the coefficients of q_i, denoted `qi coefficient` in Figure 6, consistently outperform the `MT unrolled` implementations. They are $1.13\times$, $1.31\times$ and $1.81\times$ as fast as the `MT unrolled` implementations for PRESENT, DES and AES respectively.

While our optimisations enable the polynomial approach to be consistently faster than the masked table approach, the cost remains non-negligible. Table II presents the number of clock cycles needed for the masked evaluation of the interpolating polynomial for the 3 considered S-boxes. Table II also shows the speedup obtained compared to `Poly reference` as more optimisations are added. The execution time depends a lot on the number of elements of the S-box: the execution time for the AES S-box is way higher than the one of the PRESENT S-box. Though, for all the test-cases, comparing the number of clock cycles required for the `Poly reference` and the `qi coefficients` implementations shows that the proposed optimisations achieved a total $2\times$ speedup approximately, which greatly improves the interpolating polynomial evaluation.

C. Study on AES subfunctions

In this section, we study the impact of the countermeasure more globally. We choose to measure the impact on execution time for the functions that compose the AES block cipher, as it is widely used, and as its functions perform very different operations. We use a software unprotected implementation of AES written in C, and manipulating only variables of type `uint8_t`, similar to [?]. The binary code used for this experimental evaluation is the same as the binary code considered in Section VI.

We measure the execution time overhead for all the AES subfunctions, and the ratio between the execution time with the countermeasure and without. The results are presented in Table III. The Table also gives the overhead obtained for SubBytes and KeySchedule using a mask table approach with unrolling instead of our approach. The overheads obtained for our approach vary a lot depending on the considered functions: from $\times 1.4$ to $\times 130.7$.

In order to explain the difference of the overheads, we count and categorise the instructions that manipulate secret data within each AES subfunction of our implementation. With the exception of S-box accesses, all the instructions have a moderate masking cost: typically, these instructions are linear w.r.t. the XOR, which means that such instructions are executed twice (once for each share) to compute the masked result. The S-box accesses are however much more costly because of the masked evaluation of an interpolating polynomial.

SubBytes gets the highest overhead because it contains the highest number (16) of S-box accesses. KeySchedule is the second biggest overhead because of its 4 S-box accesses. Other functions do not contain any S-box access, and as such their costs are much lower: between $\times 1.4$ and $\times 2.0$. The difference of overheads of AddRoundKey, ShiftRows, and MixColumns can be explained by the proportion of instructions to be masked in these functions.

This study highlights the high variability of the cost of the countermeasure: the masking overhead is quite low for some operations, but it is very high for others. The overall overhead of the full AES is highly impacted by the cost of the masked S-box accesses: the total overhead is $\times 32.6$. Without our optimisations though, the overhead would be $\times 64.5$ instead.

VIII. Discussion

In this section we compare our work with other approaches that leverage compilation to enforce masking [?], [?], [?].

The approach proposed by Moss *et al.* [?] is based on a type system that extends types with confidentiality information and associates an immutable list of masks to any secret variable. However, immutable lists of masks prevent the handling of control-flow structures; hence, the conversion to an intermediate representation also unrolls loops. On the contrary, in our approach, both the confidentiality and the remasking analyses deal with control flow without requiring any code transformation. Moreover, our confidentiality analysis is able to propagate the confidentiality information through loads and stores in presence of any pointer indirection to secret as well as data structures with different confidentiality levels. Finally, in [?], the confidentiality analysis is only carried out at the granularity of variables since the compiler only supports XORs and table accesses. Our confidentiality analysis combines a word-level and a bit-level analysis to support any boolean cryptographic implementation. This is essential to avoid potential false negatives which can occur, for example, with shift instructions manipulating secret data.

Agosta *et al.* [?] propose a vulnerability assessment of program instructions in symmetric ciphers. The vulnerability is quantified as the minimum number of key bits influencing the output value of a sensitive instruction. Given a vulnerability threshold, a boolean masking countermeasure is selectively applied on the hardened program such that the performance overhead is mitigated as compared to a fully masked implementation. The vulnerability analysis is achieved by data-flow analysis (forward or backward), and leverages control-flow transformations such as loop peeling and if-conversion. Loop peeling is particularly efficient as it enables to separate loop iterations where variables depend on a low number of key bits from iterations where variables depend on a high number of key bits. In our approach, we assume that any variable that depends on a secret can leak sensitive information as it would be required to not only target ciphers. Thus, our approach does not require to modify the control flow: we propose two data-flow algorithms that iterate over the code to identify the sensitive instructions to mask and to find the set of lists of masks associated to each variable. Our confidentiality analysis is similar to the forward analysis proposed in [?] while working at the variable level with uniform typing rules for dealing with loads and stores. Moreover, we detail how we transform the code and propose a fine-grained remasking analysis, whereas [?] lacks information about these steps. However, it would be possible to integrate the vulnerability assessment proposed by Agosta *et al.* in our approach, and apply the masking countermeasure accordingly, without changing neither the transformation step nor the remasking analysis.

Bayrak *et al.* also propose a confidentiality analysis and the application of boolean masking on software cryptographic implementations [?]. The code hardening is applied to an intermediate representation extracted by disassembling the machine code of the target program. The approach features two different kinds of confidentiality analysis: (1) a so-called dynamic analysis, which links measurements with program instructions to determine which instructions are the most sensitive and must be masked, and (2) a static data-flow analysis. The dynamic analysis highly depends on the measurement setup, but it is very interesting as it allows to specialise a masking counter-measure w.r.t. the intrinsic side-channel properties of the target. The static analysis is quite similar to our data-flow confidentiality analysis even though [?] does not detail if it is capable of propagating information though several memory loads and stores. The static analysis requires to reconstruct enough information from the binary program, in particular regarding secret variables. Binary analysis on optimised code is generally known as a difficult problem, since e.g. local data optimisation such as scalarisation could prevent the correct identification of secret data information. Bayrak *et al.* handle the mask collision problem differently than we do: instead of detecting when to remask, they consider a fix set of masks that they attribute to variables using an edge-colouring algorithm. This approach can reduce

TABLE III
NUMBER AND NATURE OF THE OPERATIONS MANIPULATING A SECRET WITHIN EACH OF THE AES SUBFUNCTIONS, AND OVERHEADS OBTAINED. THE OVERHEADS OBTAINED USING MASKED TABLES WITH UNROLLING INSTEAD OF MASKED POLYNOMIAL INTERPOLATION ARE ALSO GIVEN (MT).

	Nature of the operations manipulating a secret							**Original implementation**	**Masked implementation**	
AES subfunctions	xor	load	store	sign extend	shift	and immediate	S-box access	time (cycles)	time (cycles)	overhead
AddRoundKey	16	32	16	0	0	0	0	174	354	x2.0
SubBytes	0	16	16	0	0	0	16	134	17509	x130.7
SubBytes (MT)									32533	x242.8
ShiftRows	0	16	16	0	0	0	0	38	55	x1.4
MixColumns	68	16	16	16	32	16	0	260	454	x1.7
KeySchedule	17	16	16	0	0	0	4	90	4533	x50.4
KeySchedule (MT)									8327	x92.2

the whole number of masks, but it also requires a mask rotation for each masked operation. Our remasking analysis aims at conservatively avoiding remasking at each operation and future work will consider optimising the whole number of masks.

Last but not least, all these approaches use a masked table approach to mask a table lookup indexed by secret data whereas we use a polynomial interpolation approach, and we additionally use a formal verification approach to check that the binaries produced are correctly masked.

IX. CONCLUSION

We presented and detailed an approach to automatically apply first-order boolean masking in the value based leakage model, during compilation at function level without any restriction on the control-flow structure of the code. We explained how to automate the approach based on interpolating polynomial of constant lookup tables [?] and proposed several optimisations to speedup its evaluation. We implemented our approach in an LLVM pass named Maskara, at the intermediate representation level, which makes our pass target-independent. We then studied more specifically the ARM target, and highlighted the danger of back-end optimisations: the optimisation level of the instruction selection pass had to be changed to avoid introducing leakages. This motivated the use of a formal verification approach at the binary level. Results of the formal verification carried out on all the AES subfunctions showed that the binaries generated by our compiler were correctly masked w.r.t. our leakage model. Finally, execution time evaluation results showed that the interpolating polynomial approach, combined with our optimisations, enables to significantly reduce the overhead of the countermeasure for codes that have lookup table accesses indexed by a secret.

Future works will consider more complex leakage models, as well as higher-order masking, and will also evaluate to which extent the proposed optimisations may be combined with other polynomial interpolation approaches such as [?].

REFERENCES

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*. Springer, 1999.

[2] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *IEEE SP*. IEEE, 2017.

[3] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *CHES*. Springer, 2010.

[4] G. Barthe, S. Belaïd, P.-A. Fouque, and B. Grégoire, "maskVerif: a formal tool for analyzing software and hardware masked implementations." *IACR Cryptology ePrint Archive*, 2018.

[5] I. B. El Ouahma, Q. L. Meunier, K. Heydemann, and E. Encrenaz, "Side-channel robustness analysis of masked assembly codes using a symbolic approach," *JCEN*, 2019.

[6] S. T. Vu, K. Heydemann, A. de Grandmaison, and A. Cohen, "Secure delivery of program properties through optimizing compilation," in *CC*, 2020.

[7] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the cost of lazy engineering for masked software implementations," in *CARDIS*. Springer, 2014.

[8] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *CHES*. Springer, 2012.

[9] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *DAC*. IEEE, 2013.

[10] A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F.-X. Standaert, and P. Ienne, "Automatic application of power analysis countermeasures," *IEEE TC*, 2013.

[11] M. Tunstall, C. Whitnall, and E. Oswald, "Masking tables—an underestimated security risk," in *FSE*. Springer, 2013.

[12] J.-S. Coron, A. Roy, and S. Vivek, "Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures," in *CHES*. Springer, 2014.

[13] L. Goubin and J. Patarin, "Des and differential power analysis the "duplication" method," in *CHES*. Springer, 1999.

[14] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer, 2008.

[15] E. Prouff and M. Rivain, "Masking against Side-Channel Attacks: A Formal Security Proof," in *EUROCRYPT*. Springer, 2013, pp. 142–159.

[16] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Annual International Cryptology Conference*. Springer, 2003, pp. 463–481.

[17] A. Biryukov, D. Dinu, Y. Le Corre, and A. Udovenko, "Optimal first-order boolean masking for embedded iot devices," in *CARDIS*. Springer, 2017.

[18] J.-S. Coron, "Higher order masking of look-up tables," in *EUROCRYPT*. Springer, 2014.

[19] C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain, "Higher-order masking schemes for S-boxes," in *FSE*. Springer, 2012.

[20] J.-S. Coron, F. Rondepierre, and R. Zeitoun, "High order masking of look-up tables with common shares," *IACR TCHES*, 2018.

[21] M. Tang, Z. Qiu, Z. Guo, Y. Mu, X. Huang, and J.-L. Danger, "A generic table recomputation-based higher-order masking," *TCAD*, 2017.

[22] A. Roy and S. Vivek, "Analysis and improvement of the generic higher-order masking scheme of fse 2012," in *CHES*. Springer, 2013.

[23] A. Mathieu-Mahias and M. Quisquater, "Mixing additive and multiplicative masking for probing secure polynomial evaluation methods," *TCHES*, 2018.

[24] N. Belleville, K. Heydemann, D. Couroussé, T. Barry, B. Robisson, A. Seriai, and H.-P. Charles, "Automatic application of software countermeasures against physical attacks," in *Cyber-Physical Systems Security*. Springer, 2018.

[25] "Fast galois field arithmetic library in c/c++." [Online]. Available: http://web.eecs.utk.edu/~jplank/plank/papers/CS-07-593/

[26] "Small portable AES128/192/256 in C," https://github.com/kokke/tiny-AES-c, Apr. 2020.