Formal Verification of Masking Countermeasures for Arithmetic Programs

Pengfei Gao, Hongyi Xie, Pu Sun, Jun Zhang, Fu Song and Taolue Chen

Abstract—Cryptographic algorithms are widely used to protect data privacy in many aspects of daily lives from smart card to cyber-physical systems. Unfortunately, programs implementing cryptographic algorithms may be vulnerable to practical power side-channel attacks, which may infer private data via statistical analysis of the correlation between power consumptions of an electronic device and private data. To thwart these attacks, several masking schemes have been proposed, giving rise to effective countermeasures for reducing the statistical correlation between private data and power consumptions. However, programs that rely on secure masking schemes are not secure a priori. Indeed, designing effective masking programs is a labor intensive and error-prone task. Although some techniques have been proposed for formally verifying masking countermeasures and for quantifying masking strength, they are currently limited to Boolean programs and suffer from low accuracy. In this work, we propose an approach for formally verifying masking countermeasures of arithmetic programs. Our approach is more accurate for arithmetic programs and more scalable for Boolean programs comparing to the existing approaches. It is essentially a synergistic integration of type inference and model-counting based methods, armed with domain specific heuristics. The type inference system allows a fast deduction of leakage-freeness of most intermediate computations, the model-counting based methods accounts for completeness, namely, to eliminate spurious flaws, and the heuristics facilitate both type inference and model-counting based reasoning, which improve scalability and efficiency in practice. In case that the program does contain leakage, we provide a method to quantify its masking strength. A distuiguished feature of our type sytem lies in its support of compositonal reasoning when verifying programs with procedure calls, so the need of inlining procedures can be significantly reduced. We have implemented our methods in a verification tool QMVERIF which has been extensively evaluated on cryptographic benchmarks including full AES, DES and MAC-Keccak. The experimental results demonstrate the effectiveness and efficiency of our approach, especially for compositional reasoning. In particular, our tool is able to automatically prove leakage-freeness of arithmetic programs for which only manual proofs exist so far; it is also significantly faster than the state-of-the-art tools: EasyCrypt on common arithmetic programs, QMSINFER, SC Sniffer and maskVerif on Boolean programs.

1 INTRODUCTION

CRYPTOGRAPHY plays a crucial role in many aspects of our daily lives from smart card to cyber-physical systems to Internet of things, forming the backbone of security mechanisms. Modern cryptography is founded on complexity theory; it is highly non-trivial to extract private data (e.g., cryptographic keys) by directly analyzing the input-output relation of cryptographic programs. However, in practice, side-channel attacks allow an attacker to efficiently extract the private data by exploiting the statistical correlation between the private data and non-functional measurements of electronic devices, for instance, power consumption [2] and execution time [3]. Implementations of almost all major cryptographic algorithms both in software and hardware, such as DES, AES, RSA and Elliptic curves, have been successfully broken [2], [4], [5], [6], [7], [8], [9], [10], [11], [12]. As an example, consider the instruction $c = p \oplus k$ where k is a private variable and p is a non-random variable. The power consumption of a device executing $c = p \oplus k$ usually depends on the value of k, which can be exploited via power based side-channel attacks (e.g., differential power analysis [13]) to deduce the value of k.

A common countermeasure to thwart power side-channel attacks is *masking*, which has been widely used to reduce the statistical correlation between private data and power consumptions via randomization. Given a security parameter d, an order-d secret-sharing masking scheme typically splits the private data k into $(d+1)$ shares such that any subset of at most d shares is *statistically independent* of k. Computation on k is then reduced to the one based on its $(d+1)$ shares. For instance, the private data k can be masked by computing the exclusive-or operation ($\oplus$) with a uniform random variable r, so-called *Boolean masking scheme* [14], leading to two shares $k \oplus r$ and r. One can observe that the probability distributions of r and $k \oplus r$ do not rely upon k. The value of k can be recovered by computing $(k \oplus r) \oplus r = k$, which is usually referred to as de-masking.

Apart from Boolean masking, arithmetic masking schemes such as additive masking schemes (e.g., $(k+r) \bmod$

- *P. Gao is with School of Information Science and Technology, ShanghaiTech University, Pudong, Shanghai, China; Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, Shanghai, China; University of Chinese Academy of Sciences, Beijing, China.*
- *H. Xie, P. Sun, J. Zhang and F. Song are with School of Information Science and Technology, ShanghaiTech University, Pudong, Shanghai, China. Fu Song is also with Shanghai Engineering Research Center of Intelligent Vision and Imaging, Shanghai, China.*
- *T. Chen is with Department of Computer Science, University of Surrey, GU2 7XH, Guildford, UK. He is also with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.*
- *Corresponding author: Fu Song*

A preliminary version of this paper [1] appeared in the Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS'19), Prague, Czech Republic, April 6-11, 2019.
Manuscript received XXX XX, 20XX; revised XXX XX, 20XX.

n) and multiplicative masking schemes (e.g., $(k \times r) \bmod n$) have also been proposed [15], [16], [17], [18], [19]. Boolean masking is adopted for algorithms that have Boolean operations only. It can be advantageous to use arithmetic masking to protect arithmetic operations. For masking cryptographic algorithms that embrace both Boolean and arithmetic operations such as IDEA [20], RC6 [21], and SPECK [22], one may need to switch between Boolean and arithmetic masking whenever necessary.

Several secure conversion algorithms between Boolean and arithmetic maskings (e.g., [16], [17], [18], [19], [23]) as well as masked programs of cryptographic algorithms (e.g., [14], [24], [25], [26], [27], [28], [29], [30], [31]) have been published over the past years. However, it is labor-intensive and error-prone to develop effective and/or efficient masked implementations particularly for non-linear functions which are widely used in cryptographic algorithms. For instance, the masked AES programs proposed by Schramm and Paar [27] is shown to be vulnerable [32], [33]. One commonly accepted remedy is to formally and automatically verify masking countermeasures of program implementations of cryptographic algorithms, which is the main topic of the current work.

Techniques for formally verifying masking countermeasures of cryptographic programs do exist. In general, these techniques can be classified into two categories: rule based approaches [34], [35], [36], [37], [38], [39], [40] and model-counting based approaches [41], [42], [43], [44]. In a nutshell, rule based approaches check the security of intermediate computation results via their syntactic information, from which one may prove leakage-freeness of the target program, or identify potential flaws. These approaches are usually sound and efficient for programs using Boolean masking schemes when the computations are syntactically independent of the private data or masked by a unique random variable. However, they are *not* complete, namely, leakage-free programs may fail to pass the verification (i.e., false positive), and spurious flaws are hard to be automatically identified so tedious manual examination is usually necessary. In contrast, model-counting based approaches reduce the verification problem to the satisfiability problem of a series of constraints which encode model-counting and are solved by leveraging SAT/SMT solvers. These approaches enjoy both soundness and completeness. However, due to the inherent complexity of the model-counting problem and the exponential blow-up induced by the reduction, these approaches pose great challenges to scalability and can be very slow in practice. Currently they are limited to *Boolean programs only*. In general, there is a shortage of verification approaches and tools that can effectively and efficiently verify masking countermeasures of *arithmetic* programs.

To tackle this problem, one naive solution is to transform arithmetic programs into equivalent Boolean programs through bit-blasting [45] and then apply existing verification tools on the Boolean programs. It is possible in principle, but practically unfavourable due to the following deficiencies: (1) arithmetic programs admit rich operations and one has to encode them (e.g., finite-field multiplication) as bitwise operations; (2) verifying the order-d security of a 8-bit program must be done by verifying the order-8^d security of its Boolean translation, where each 8^d-tuple of internal variables in the Boolean translation corresponds to a d-tuple of internal variables in the original 8-bit program. This means that verifying a first-order 8-bit program with m internal variables must be done by performing m verifications on sets of 8 Boolean variables such that each set corresponds to an internal variable in the 8-bit program. Note that the state-of-the-art higher-order verification tool maskVerif [37] already takes more than 18 minutes to verify just order-5 masked Boolean implementation of DOM Keccak Sbox [46] which only contains 618 internal variables.

In this article, we propose an approach for formally verifying the security of first-order masking countermeasures of *arithmetic* programs without bit-blasting. Essentially, our approach is a synergistic integration of type systems and model-counting based methods. We introduce a new type system for inferring distribution types of internal variables by designing inference rules for both Boolean and arithmetic operations. It is often able to quickly obtain soundness proofs when the program is leakage-free. A distinguished feature of the type system lies in its support for compositional reasoning so inlining procedures in the program can be largely avoided or be reduced at least. To resolve problems that cannot be proved by the type system, we propose two model-counting based methods: a brute-force method and an SMT-based method. The brute-force method computes the probability distribution of a potential flaw by exhaustively enumerating all possible valuations of variables. The SMT-based method transforms the verification problem of a potential flaw to the satisfiability problem of a (quantified-free) first-order logic formula that can be solved by SMT solvers (e.g. Z3 [47]). Although expensive, model-counting based methods are powerful to completely determine if the potential flaw is spurious or not. Furthermore, we propose three heuristics to simplify the intermediate computations of internal variables. These heuristics allow the type system to resolve more inclusive answers and thus reduce the burden of model-counting, which could significantly improve the scalability and efficiency of our approach.

Perfect masking is ideal, but does not necessarily hold in practice. In certain scenarios, there are intended flaws when only a limited number of random variables are allowed for efficiency consideration [48]. However, when this is the case, it is important to measure the resource the attacker needs in order to infer the private data via power side-channels. For this purpose, we adapt the notion of *Quantitative Masking Strength* (QMS), which was proposed by Eldib et al. [49], [50]. It is empirically shown that there is a correlation between the number of power traces to successfully infer private data and QMS values [49], [50]. We propose a binary search based algorithm to compute QMS values of flaws in Boolean/arithmetic programs by leveraging model-counting based methods. We remark that the approach of Eldib et al. [49], [50] approximates QMS values on Boolean programs only.

We have implemented our approach in a verification tool QMVERIF (**Q**uantitative **M**asking **V**ERIFier) and conducted extensive experiments on masked Boolean and arithmetic programs including the full AES, DES and MAC-Keccak implementations. QMVERIF could be used to verify high-level arithmetic programs at design and implementation stages, when these programs are supposed be deployed in security-critical software, especially when their power consumptions of the execution may be probed by attackers.

Contributions. We summarize the main contributions as follows.

- We propose a type system supporting compositional reasoning, two model-counting based methods, and their synergistic integration with domain specific heuristics, which can efficiently and effectively prove masking countermeasures for both Boolean and arithmetic programs; the approach is not only sound but also complete.
- We propose a binary search based algorithm for computing exact quantitative masking strength of arithmetic programs by leveraging our model-counting based methods.
- We develop an open-source software tool that implements the above approaches and heuristics for a specifically designed language. It supports both qualitative and quantitative verification of masking countermeasures of Boolean and arithmetic programs.
- We conduct extensive experiments on both masked Boolean and arithmetic programs including full AES, DES and MAC-Keccak implementations. Experimental results demonstrate the effectiveness of our approach, and show orders of magnitude improvement with respect to previous verification methods on common benchmarks.

It is worth mentioning that our approach and tool can automatically prove the security of several conversion algorithms (e.g., implementations of Boolean to arithmetic masking [16], [17], [19] and arithmetic to Boolean masking [16], [17]). To the best of our knowledge, it is the first time that they are proved leakage-free by computer-aided tools rather than manually.

One feature of our approach is that it could avoid inlining procedure calls in the program via supporting compositional reasoning in the assume-guarantee style. The experiments show it is able to verify various implementations of Sbox and full AES in less than one second when procedure assumptions are provided. Even when no procedure assumptions or only one procedure assumption is provided, lots of procedure inlines can be avoided. Our experiments also find, perhaps surprisingly, that for solving model-counting constraints, the widely adopted methods based on SMT solvers (e.g. [41], [42], [43], [44]) may not be the best option, as the alternative brute-force method is comparable for Boolean programs, and significantly faster for arithmetic programs with (finite-field) multiplication, hence calls for further effort towards the solving of domain-specific model-counting constraints.

This paper is an extension of the conference paper [1], and is related to our previous work [43], [44]. Detailed comparison between them are given in Section 6.

Organization. The rest of the paper is organized as follows. In Section 2, we introduce cryptographic programs considered in this work, leakage and threat models, and the notions of perfect masking and quantitative masking strength. Section 3 gives a running example used to illustrate our techniques and an overview of our approach. Section 4 presents our methodology, including a type system supporting compositional reasoning (Section 4.1), two model-counting based methods (Section 4.2). three heuristics to improve scalability and efficiency in practice (Section 4.3) and the overall algorithms (Section 4.4). Section 5 reports experimental results. We discuss related work in Section 6. Finally, we conclude the work in Section 7.

The implementation of QMVERIF is open-sourced, available at http://s3l.shanghaitech.edu.cn/software/qmverif.

2 PRELIMINARIES

In this section, we introduce the cryptographic programs which will be considered in this article, threat model and leakage models, as well as the notions of perfect masking and quantitative masking strength.

We fix a natural number $n > 0$ and an integer domain $\mathfrak{D} = \{0, \cdots, 2^n - 1\}$. The domain $\mathfrak{D}$ is isomorphic to the Galois field $\mathbb{GF}(2)[x]/(p(x))$ (or simply $\mathbb{GF}(2^n)$) for some irreducible polynomial p, e.g., $\mathbb{GF}(2^8)$ and $p(x) = x^8 + x^4 + x^3 + x + 1$, which is usually referred to as Rijndael's (AES) finite field. We will denote by $\vec{1}$ the value $2^n - 1 \in \mathfrak{D}$.

2.1 Cryptographic Programs

In this article, we consider cryptographic programs rather than arbitrary software programs. It is common to assume that cryptographic programs are branching-free (i.e., in straight-line forms) for formal verification [35], [42]. (Remark that our tool supports programs with static loops by loop unfolding. We do not consider cryptographic programs that inherently contain branching in this work, but some programs which can be transformed to the branching-free form can be tackled.)

Syntax. The syntax of the program under verification is given in Figure 1. A (cryptographic) program P consists of a sequence of procedure definitions $f(a_1, \cdots, a_m)$, where f denotes the procedure name and $a_1, \cdots, a_m$ are the formal arguments of f. We assume that the procedure names of P are distinct, there is a unique procedure named *main* as the entry point of P, and all the procedures only use local variables and formal arguments, but no global variable unless the program contains only the *main* procedure. A procedure $f(a_1, \cdots, a_m)$ consists of a sequence of assignments followed by a return statement `return` $x_1, \cdots, x_k$. Note that a return statement could return more than one value in our language for the sake of convenience.

An assignment of the form $x = e$, as usual, assigns the value of the expression e to the variable x. An assignment of the form $r = \$$ assigns a uniformly sampled random value from the domain $\mathfrak{D}$ to the variable r where effectively r is a random variable. An assignment of the form $x_1, \cdots, x_k = f(y_1, \cdots, y_m)$ is a procedure call which passes the actual arguments $y_1, \cdots, y_m$ to the formal arguments $a_1, \cdots, a_m$ of f, executes the function body of $f(a_1, \cdots, a_m)$ and finally assigns the return values to the variables $x_1, \cdots, x_k$, assuming that the number of return values of f is k.

We assume that each procedure call $x_1, \cdots, x_k = f(y_1, \cdots, y_m)$ is associated with a unique call-site ℓ (e.g.,

Constant:	$\mathfrak{D} \ni c$	::=	n-bit constant
Operation:	$\mathcal{O} \ni \circ$	::=	$\wedge \mid \vee \mid \oplus \mid - \mid + \mid \times \mid \odot$
Expression:	e	::=	$c \mid x \mid \neg e \mid e \ll c \mid e \gg c \mid e \circ e$
Statememt:	`stmt`	::=	$x = e \mid r = \$ \mid x_1, \cdots, x_k = f(y_1, \cdots, y_m) \mid$ `stmt;stmt`
Procedure:	$F \ni f(a_1, \cdots, a_m)$	::=	`stmt;return` $x_1, \cdots, x_k$;
Program:	P	::=	F^+

Fig. 1. Syntax of the programming language used by QMVerif

line number) and let $f(y_1, \cdots, y_m)[i]@\ell$ denote the i-th return value of the procedure call $f(y_1, \cdots, y_m)$ at the call-site ℓ. Therefore, the procedure call $x_1, \cdots, x_k = f(y_1, \cdots, y_m)$ can be treated as a sequence of assignments:

$$\begin{aligned} x_1 &= f(y_1, \cdots, y_m)[1]@\ell; \\ &\cdots; \\ x_k &= f(y_1, \cdots, y_m)[k]@\ell;. \end{aligned}$$

An expression e is built up from n-bit variables and constants using the following operations:

- **bit-wise operations**: *and* ($\wedge$), *or* ($\vee$), *negation* ($\neg$), *exclusive-or* ($\oplus$), *right shift* $\gg$ and *left shift* $\ll$;
- **modulo 2^n arithmetic operations**: *subtraction* ($-$), *addition* ($+$), and *multiplication* ($\times$), for which $\mathfrak{D}$ is considered to be $\mathbb{Z}_{2^n}$, i.e, the ring of integers modulo 2^n;
- **finite-field operation**: *multiplication* ($\odot$), for which $\mathfrak{D}$ is considered to be a Galois field $\mathbb{GF}(2^n)$. (Note that addition and subtraction operations over Galois fields are essentially bit-wise exclusive-or.)

In the rest of the paper, we denote by $\mathcal{O}^*$ the set of operations $\mathcal{O} \cup \{\ll, \gg\}$. For each procedure $f(a_1, \cdots, a_m)$ defined in the program P, let X^f denote the set of variables defined in the procedure $f(a_1, \cdots, a_m)$ (called internal variables), $X^f_r \subseteq X^f$ denote the set of random variables defined in the procedure $f(a_1, \cdots, a_m)$, and X^f_a denote the set of formal arguments $\{a_1, \cdots, a_m\}$. We assume, without loss of generality, that each variable $x \in X^f$ is defined at most once in $f(a_1, \cdots, a_m)$, namely, the procedure $f(a_1, \cdots, a_m)$ is in the *single static assignment* (SSA) form, and each expression contains at most one operation. Indeed, any straight-line procedure can be transformed into the one satisfying these conditions. For the main procedure $main(a_1, \cdots, a_m)$, the set of formal arguments X^{main}_a is partitioned into two disjoint sets: public input variables (X_p) and private input variables (X_k).

Computation. For each variable x used in the procedure $f(a_1, \cdots, a_m)$, the (intermediate) *partial computation* $\mathcal{E}(x)$ of x is defined as follows:

- If x is a formal argument or random variable, i.e., $x \in X^f_a \cup X^f_r$, then $\mathcal{E}(x) = x$;
- Otherwise, $\mathcal{E}(x)$ is obtained by
  1) initially, $\mathcal{E}(x) = e$ if x is defined by the assignment $x = e$, or $\mathcal{E}(x) = g(y_1, \cdots, y_m)[i]@\ell$ if the procedure call $x_1, \cdots, x_{i-1}, x, x_{i+1}, \cdots, x_k = g(y_1, \cdots, y_m)$ is made at the call-site ℓ in $f(a_1, \cdots, a_m)$;
  2) then recursively replacing each variable y in $\mathcal{E}(x)$ with its partial computation $\mathcal{E}(y)$ until the updating is stabilized.

Intuitively, the partial computation $\mathcal{E}(x)$ of x is an expression in terms of random variables (X^f_r) and formal arguments (X^f_a), *without* inlining procedure calls. Being in the single static assignment form guarantees that $\mathcal{E}(x)$ is well-defined.

A partial computation $\mathcal{E}(x)$ is a *full computation* if $\mathcal{E}(x)$ does not contain any procedure calls and all the formal arguments used in $\mathcal{E}(x)$ are from X^{main}_a (i.e., the formal arguments of the *main* procedure).

Procedure inlining. In this paper, we consider non-recursive programs, for which we can inline all the procedure calls so that the resulting program contains only the *main* procedure. For the sake of presentation, we introduce the procedure inlining as follows.

For each procedure call $x_1, \cdots, x_k = f(y_1, \cdots, y_m)$ at the call-site ℓ in the procedure g where the procedure body of f is

$$f(a_1, \cdots, a_m) = s_1; \cdots s_t; \texttt{return}\ z_1, \cdots, z_k;,$$

we inline the procedure call $x_1, \cdots, x_k = f(y_1, \cdots, y_m)$ by replacing them with the following statements:

$$\begin{aligned} & a_1@\ell = y_1;\ \cdots;\ a_m@\ell = y_m; \\ & s'_1;\ \cdots\ s'_t; \\ & x_1 = z_1@\ell;\ \cdots\ x_k = z_k@\ell; \end{aligned}$$

where for every $1 \leq i \leq t$, the statement s'_i denotes the statement obtained from s_i by replacing every variable $z \in X^f \cup X^f_a$ with $z@\ell$. Moreover, if s_i is a procedure call with the call-site ℓ', then the call-site of s'_i becomes $\ell'@\ell$ which tracks the call-site ℓ. It follows that a call-site ℓ may be a sequence of call-sites of the form $\ell_k@\cdots@\ell_1$. The resulting procedure of g is denoted by $\texttt{inline}(g, \ell)$, namely, the procedure call at the call-site ℓ in g is inlined. For a sequence of procedure calls with call-sites $\ell_1, \cdots, \ell_k$, we denote by $\texttt{inline}(g, \ell_1, \cdots, \ell_k)$ the procedure $\texttt{inline}(...\texttt{inline}(\texttt{inline}(g, \ell_1), \ell_2@\ell_1), ..., \ell_k@\cdots@\ell_1)$, with $\texttt{inline}(g, \ell_1, \cdots, \ell_k) = g$ if $k = 0$.

For any non-recursive program P, by iteratively inlining all the procedure calls, we can obtain an equivalent program, denoted by $P_{\texttt{inlined}}$, which only has the *main* procedure. Assuming that the variable names used in the program P do not contain @, the program $P_{\texttt{inlined}}$ is in the SSA form. For a variable $x \in X^f$ defined in a procedure $f(a_1, \cdots, a_m)$, x will become the variables $x@\ell_k \cdots @\ell_1$ in $P_{\texttt{inlined}}$, for sequences of call-sites $\ell_1 \cdots \ell_k$ from the procedure *main* to the procedure f in the call graph of P. We denote by $\texttt{inline}(x)$ the set of such variables in $P_{\texttt{inlined}}$. Obviously, each internal variable x defined in $P_{\texttt{inlined}}$ has a unique full computation $\mathcal{E}(x)$. Moreover, a partial computation $\mathcal{E}(x)$ defined in the procedure $f(a_1, \cdots, a_m)$ corresponds to the full computations $\mathcal{E}(x')$ for the variables $x' \in \texttt{inline}(x)$ in $P_{\texttt{inlined}}$.

Similarly, for any partial computation $\mathcal{E}(x)$ of a variable x defined in a procedure g, and a procedure call $f(e_1, \cdots, e_m)$ at the call-site ℓ in the procedure g, all the terms of the form $f(e_1, \cdots, e_m)[i]@\ell$ in $\mathcal{E}(x)$ can be inlined by replacing it with the partial computation $\mathcal{E}'(z_i)$, where $\mathcal{E}'(z_i)$ is obtained from $\mathcal{E}(z_i)$ of the procedure body

$$f(a_1, \cdots, a_m) = s_1; \cdots s_t; \texttt{return}\ z_1, \cdots, z_k;$$

by replacing the formal arguments $a_1, \cdots, a_m$ in $\mathcal{E}(z_i)$ with the partial computations $e_1, \cdots, e_m$ respectively, replacing random variables r in $\mathcal{E}(z_i)$ by $r@\ell$, and replacing the symbol $@\ell'$ in $\mathcal{E}(z_i)$ with $@\ell'@\ell$. Indeed, the resulting partial computation, denoted by $\texttt{inline}(\mathcal{E}(x), \ell)$, is the partial computation $\mathcal{E}(x)$ of the variable x in the procedure $\texttt{inline}(g, \ell)$. We denote by $\mathcal{E}(x)_{\texttt{inlined}}$ the partial computation of the variable x obtained by iteratively inlining all the terms of the form $f(e_1, \cdots, e_m)[i]@\ell$. When x is a variable defined in the *main* procedure, i.e., $x \in X^{main}$, $\mathcal{E}(x)_{\texttt{inlined}}$ is a full computation of x in the program $P_{\texttt{inlined}}$.

Semantics. A *valuation* for a set of variables Y is a function assigning to each variable $y \in Y$ a concrete value $c \in \mathfrak{D}$. For a subset of variables $Z \subseteq Y$, two valuations (σ_1, σ_2) are *Z-equivalent*, denoted by $\sigma_1 \simeq_Z \sigma_2$, if $\sigma_1(z) = \sigma_2(z)$ for all variables $z \in Z$. We denote by Θ the set of valuations for the set of variables $X_p \cup X_k$. Given an expression (i.e., computation) e and a valuation $\sigma \in \Theta$, let $e(\sigma)$ be the expression obtained from e in which all the variables $x \in X_p \cup X_k$ are instantiated by the concrete values $\sigma(x)$. By abuse of notation, for an assignment σ of formal arguments and variables in the partial computation e, we also denote by $e(\sigma)$ the expression obtained from e, where all the formal arguments and variables x in e are instantiated by the concrete values $\sigma(x)$.

For a full computation e, the random variables in $e(\sigma)$ are uniformly distributed. We write $[\![e]\!]_\sigma$ for the resulting random variable which gives rise to a distribution as follows. Namely, for each concrete value $c \in \mathfrak{D}$,

$$[\![e]\!]_\sigma(c) = \frac{|\{\mu : X_r \to \mathfrak{D} \mid e(\sigma, \mu) = c\}|}{|\mathfrak{D}|^{|X_r|}}.$$

where $e(\sigma, \mu)$ denotes the value of the full computation $e(\sigma)$ by instantiating random variables $r \in X_r$ with concrete values $\mu(r)$. As a result, $[\![e]\!]_\sigma(c)$ is the probability that $e(\sigma)$ evaluates to c under the valuation σ.

Given a program P, for each variable $x \in X^{main}$ of the program $P_{\texttt{inlined}}$ and valuation $\sigma \in \Theta$, we denote by $[\![x]\!]_\sigma$ the distribution $[\![\mathcal{E}(x)]\!]_\sigma$ (note that $\mathcal{E}(x)$ must be a full computation). The *semantics* of P is a (partial) function $[\![P]\!]$ that gives the distribution $[\![x]\!]_\sigma$ for each valuation $\sigma \in \Theta$ and variable $x \in X^{main}$ of the program $P_{\texttt{inlined}}$.

2.2 Threat Model and Leakage Models

In this work, we adopt a commonly used threat model [41], [42], [43], [44], [51], [52], which assumes that the adversary has access to public input variables X_p, but not to private input variables X_k, of the program P. Moreover, the adversary may have access to results of intermediate full computations (i.e., internal variable x in $P_{\texttt{inlined}}$) via power side-channel information. Under these assumptions, the goal of the adversary is to deduce the information of X_k.

For power side-channel attacks, it is the correlations between power consumption values, rather than the absolute power consumption, that matters. The correlation between power consumption values usually comes from, for instance, the leakage currents of CMOS transistors which comprise static and dynamic leakage currents. The former always exists, but its volume depends on whether the transistor is on or off which corresponds to the logical 1 and 0 of a bit. The latter occurs only when a transistor is switched (bit flip) which corresponds to the switch between logical 1 and 0 of a bit. Both static and dynamic leakage currents can be used by the adversary to infer the private data.

Towards formally verifying masking countermeasures, we define a leakage model that precisely captures the information that may be leaked to the adversary. In this work, we consider two such models: the *Hamming Weight* (HW) and *Hamming Distance* (HD) leakage models. Both models have been used as leakage models for verifying masking countermeasures [40], [41], [42], [43], [52], and been validated on real devices [2], [6], [13], [53], [54], [55].

2.2.1 HW Leakage Model

The HW leakage model maps intermediate full computation results (i.e., data values) of an executing program to power consumptions that are induced by static leakage currents. For a constant $c \in \mathfrak{D}$, the *Hamming weight* of c, denoted by $\mathsf{HW}(c)$, is the number of logical 1 bits in c, namely,

$$\mathsf{HW}(c) = \Sigma_{i=0}^{n-1} c_i,$$

where c_i denotes the i-th greatest significant bit of c. Intuitively, $\mathsf{HW}(c)$ measures the power consumptions of CMOS transistors (e.g., register) for storing the constant c. For instance, consider the instruction $\texttt{a} = \texttt{0xFF} \oplus \texttt{k}$ where k is a private input variable. If $\texttt{k} = \texttt{0x00}$ (resp. $\texttt{k} = \texttt{0xFF}$), then the value of $\texttt{a}$ is $\texttt{0xFF}$ (resp. $\texttt{0x00}$). The power consumption of executing this instruction is proportional to $\mathsf{HW}(\texttt{0xFF}) = 8$ (resp. $\mathsf{HW}(\texttt{0x00}) = 0$), hence depends on the value of k. The adversary can infer the value of k via attacks that use the HW leakage model such as the simple power analysis in [56] or the differential power analysis in [2], [57].

2.2.2 HD Leakage Model

The HD leakage model maps intermediate full computation results of an executing program to power consumptions that are induced by dynamic leakage currents. For two constants $c, c' \in \mathfrak{D}$, the *Hamming distance* of c and c' that are consecutively assigned to a variable, denoted by $\mathsf{HD}(c, c')$, is the number of positions at which the logical values are different at c and c'. Namely,

$$\mathsf{HD}(c, c') = \Sigma_{i=0}^{n-1}(c_i \oplus c'_i) = \mathsf{HW}(c \oplus c').$$

Intuitively, for two constants $c, c' \in \mathfrak{D}$ that are consecutively assigned to the same variable, $\mathsf{HD}(c, c')$ measures the power consumptions of CMOS transistors (e.g., register) that update from c to c' via switching between logical 1 and 0. For instance, consider two instructions

$$\texttt{a} = \texttt{r}_1 \oplus \texttt{r}_2;$$
$$\texttt{a} = \texttt{a} \oplus \texttt{k};$$

where r_1, r_2 are two random variables, and k is a private input variable. For any value of k, the Hamming weights of the values of a are uniformly distributed, so the adversary cannot infer the value of k via attacks using the HW leakage model, e.g., the simple power analysis in [56] or the differential power analysis in [2], [57] that use the HW leakage model. However, the Hamming distance $\mathsf{HD}(c, c')$ of two consecutive values c, c' of a is the Hamming weight of the value of k, i.e., $\mathsf{HD}(r_1 \oplus r_2, (r_1 \oplus r_2) \oplus k) = \mathsf{HW}(k)$. Therefore, the adversary is able to infer the value of k via attacks that use the HD leakage model, e.g., the correlation power analysis in [54]. Note that the simple power analysis [56] or the differential power analysis [2], [57] could be used to infer the value of k, if the HD leakage model is used. Similarly, the correlation power analysis [54] could be adapted for the HW leakage model. The details of power consumption, HW/HD leakage models and their relation are given in, e.g., [13].

We remark that the HW leakage model is equivalent to the (value-based) first-order probing model proposed by Ishai et al. [14], and the HD leakage model is equivalent to the transition-based first-order probing model [58], but the HD leakage model differs from the (value-based) second-order probing model as shown by Wang et al. [40].

2.2.3 From HD Leakage Model to HW Leakage Model

Since we assume that each variable is defined at most once, i.e., no variables will be assigned twice, hence in theory no leakage occurs under the HD leakage model. However, in practice, some values may be assigned to the same variable in the original programs (i.e., programs before being transformed to SSA forms) or in low-level programs due to register allocation and assignment. To alleviate this problem, we assume that, when the HD leakage model is considered, a set of variable pairs is associated with each procedure of the SSA program. Intuitively, the variables in each pair (x_1, x_2) refer to the same variable x in the original program or in the low-level program after register allocation and assignment, and they are used to record, for instance, two consecutive values of x before and after assignment (as intermediate computation results of $\mathcal{E}(x_1)$ and $\mathcal{E}(x_2)$). If one wants to consider pairs of variables from different procedures, the program P can be transformed into an equivalent program $P_{\texttt{inlined}}$ and verify $P_{\texttt{inlined}}$ under the HD leakage model.

The set of variable pairs could be obtained by inspecting the transformation from the original program to the SSA form or register allocation and assignment. To verify the program under the HD leakage model, we reduce to verifying a new program under the HW leakage model by (1) adding a dummy variable $x_{1,2}$ for each variable pair (x_1, x_2) and (2) inserting a new instruction $x_{1,2} = x_1 \oplus x_2$ after the assignments of x_1 and x_2, as $\mathsf{HD}(x_1, x_2) = \mathsf{HW}(x_1 \oplus x_2) = \mathsf{HW}(x_{1,2})$. Therefore, for ease of presentation, we shall use the HW leakage model during the illustration of our approach.

We remark that our formal verification approach is general and could be integrated into compilation as done by Wang et al. [40] so that the set of variable pairs could be automatically inferred.

2.3 Perfect Masking

We fix a program P in this section. For each internal variable x of the program $P_{\texttt{inlined}}$, we say x is *uniform* in $P_{\texttt{inlined}}$, denoted by x-**UF**, if $[\![x]\!]_\sigma$ is a uniform distribution for all valuations $\sigma \in \Theta$, and x is *statistically independent* in $P_{\texttt{inlined}}$, denoted by x-**SI**, if $[\![x]\!]_{\sigma_1} = [\![x]\!]_{\sigma_2}$ for all pairs of valuations $(\sigma_1, \sigma_2) \in \Theta^2_{X_p}$, where $\Theta^2_{X_p}$ denotes the set $\{(\sigma_1, \sigma_2) \in \Theta \times \Theta \mid \sigma_1 \simeq_{X_p} \sigma_2\}$. It is straightforward to see that if $P_{\texttt{inlined}}$ is x-**UF**, then $P_{\texttt{inlined}}$ is x-**SI**. Note that the inverse does not hold in general.

An internal variable x of $P_{\texttt{inlined}}$ is called *perfectly masked* in $P_{\texttt{inlined}}$ if it is x-**SI**, otherwise x is called leaky. The program $P_{\texttt{inlined}}$ is *perfectly masked* if all internal variables in $P_{\texttt{inlined}}$ are perfectly masked. Intuitively, if the program $P_{\texttt{inlined}}$ is x-**UF**, then the values of x for each valuation $\sigma \in \Theta$ are uniformly distributed. This implies that the Hamming weights of the values of x, hence the power consumptions, are uniformly distributed. Therefore, the adversary cannot deduce any information of private data through the variable x. Note that a difference between distributions which does not result in a difference under the HW model can still be used for an attack, so we define perfect masking in the above form. Similarly, if the program $P_{\texttt{inlined}}$ is x-**SI**, then the distributions of values of x for each pair of valuations $(\sigma_1, \sigma_2) \in \Theta^2_{X_p}$ are the same. This implies that the distributions of Hamming weights of the values of x are the same. Therefore, the distributions of power consumptions through the variable x do not rely on private data and the adversary cannot deduce any information of private data through the variable x. We say the program P is *perfectly masked* if the program $P_{\texttt{inlined}}$ is perfectly masked.

To verify whether the program P is leakage-free, we focus on the leaks of individual internal variables of $P_{\texttt{inlined}}$ instead of the whole program $P_{\texttt{inlined}}$. If all the individual internal variables of $P_{\texttt{inlined}}$ are leakage-free, i.e., the program $P_{\texttt{inlined}}$ is x-**SI** for all internal variables x of $P_{\texttt{inlined}}$, then the whole programs P and $P_{\texttt{inlined}}$ are leakage-free, i.e., private data in the program P is perfectly masked.

As an example, consider a program snippet P shown below, where k_0, k_1 are private variables, and r_0, r_1 are random variables. P is x_0-**UF** and x_1-**UF**, but x_2 is leaky, as the value of x_2 statistically depends on k_1.

Program P	Modified Program P'
$x_0 = r_0 \oplus k_0;$	$x_0 = r_0 \oplus k_0;$
$x_1 = r_0 \oplus k_1;$	$x_1 = r_0 \oplus k_1;$
$x_2 = r_1 \wedge k_1;$	$x_{0,1} = x_0 \oplus x_1;$
	$x_2 = r_1 \wedge k_1;$

Suppose the same register is assigned to x_0 and x_1, then the security of P under the HD leakage model can be checked by verifying the modified program P' under the HW leakage model. Since $x_{0,1}$ is leaky in P' under the HW leakage model and $\mathsf{HW}(x_{0,1}) = \mathsf{HD}(x_0, x_1) = \mathsf{HW}(r_0 \oplus k_0 \oplus r_0 \oplus k_1) = \mathsf{HW}(k_0 \oplus k_1)$, we deduce that P under the HD leakage model is not secure.

2.4 Quantitative Masking Strength

To quantify masking strength of Boolean programs, Eldib et al. [49], [50] introduced a notion, called *Quantitative Masking*

```
1  SecMult($a_0, a_1, b_0, b_1$){        //$a_0 \oplus a_1 = a, b_0 \oplus b_1 = b$
2    $r_0 = \$$;
3    $x_0 = a_1 \odot b_0$;
4    $x_1 = a_0 \odot b_1$;
5    $x_2 = x_1 \oplus r_0$;
6    $x_3 = x_2 \oplus x_0$;
7    $x_4 = a_0 \odot b_0$;
8    $x_5 = x_4 \oplus r_0$;
9    $x_6 = a_1 \odot b_1$;
10   $x_7 = x_6 \oplus x_3$;
11   return $x_5, x_7$;                   // $x_5 \oplus x_7 = a \odot b$
12 }
13
14 RefreshMasks($a_0, a_1$){              //$a_0 \oplus a_1 = a$
15   $r_1 = \$$;
16   $y_0 = a_0 \oplus r_1$;
17   $y_1 = a_1 \oplus r_1$;
18   return $y_0, y_1$;                   // $y_0 \oplus y_1 = a$
19 }
20
21 SecExp254($a_0, a_1$){                 //$a_0 \oplus a_1 = a$
22   $z_0 = a_0^2$;
23   $z_1 = a_1^2$;                                  //$z_0 \oplus z_1 = a^2$
24   $z_2, z_3$ =RefreshMasks($z_0, z_1$);  //$z_2 \oplus z_3 = a^2$
25   $z_4, z_5$ =SecMult($z_2, z_3, a_0, a_1$);  //$z_4 \oplus z_5 = a^3$
26   $z_6 = z_4^4$;
27   $z_7 = z_5^4$;                                  //$z_6 \oplus z_7 = a^{12}$
28   $z_8, z_9$ =RefreshMasks($z_6, z_7$);  //$z_8 \oplus z_9 = a^{12}$
29   $z_{10}, z_{11}$ =SecMult($z_4, z_5, z_8, z_9$); //$z_{10} \oplus z_{11} = a^{15}$
30   $z_{12} = z_{10}^{16}$;
31   $z_{13} = z_{11}^{16}$;                         //$z_{12} \oplus z_{13} = a^{240}$
32   $z_{14}, z_{15}$ =SecMult($z_{12}, z_{13}, z_8, z_9$); //$z_{14} \oplus z_{15} = a^{252}$
33   $z_{16}, z_{17}$ =SecMult($z_{14}, z_{15}, z_2, z_3$); //$z_{16} \oplus z_{17} = a^{254}$
34   return $z_{16}, z_{17}$;
35 }
36
37 main($k$){
38   $r = \$$;
39   $k_1 = r \oplus k$;
40   $k_2, k_3$ =SecExp254($r, k_1$);  //$k_2 \oplus k_3 = k^{254}$
41 }
```

Fig. 2. The program P^{254} that implements the first-order secure exponentiation to the power 254 over $\mathbb{GF}(2^8)$ [28].

Strength (QMS), which is a generalization of perfect masking. It was empirically shown that, for Boolean programs the number of power traces needed to successfully infer the private data from the computation results of an internal variable x is exponential in the QMS value of x.

In this work, we generalize the notion of QMS from Boolean setting to the arithmetic one. The *Quantitative Masking Strength* (QMS) of an internal variable x of $P_{\texttt{inlined}}$ is defined as:

$$\texttt{QMS}_x := 1 - \max_{\substack{(\sigma_1,\sigma_2)\in\Theta^2_{X_p} \\ c\in\mathfrak{D}}} \Big([\![x]\!]_{\sigma_1}(c) - [\![x]\!]_{\sigma_2}(c) \Big).$$

It is easy to see that $P_{\texttt{inlined}}$ is x-**SI** iff $\texttt{QMS}_x = 1$. We remark that the notion of QMS is same as the one in Eldib et al. [49], [50] when $n = 1$, i.e., the domain $\mathfrak{D}$ becomes the Boolean domain $\{0, 1\}$.

Research objective. The main goal of this work is to verify whether a cryptographic program P is perfectly masked, and to assess how strong it is for each leaky variable in terms of QMS in case that P is not perfectly masked.

3 Running Example and Overview

In this section, we present a running example and an overview of our approach.

3.1 A Running Example

We illustrate the notions and techniques by the program P^{254} shown in Figure 2, which implements the first-order secure exponentiation to the power 254 over $\mathbb{GF}(2^8)$ [28], i.e., computes k^{254} for a given input k. To thwart first-order side-channel attacks, the private input variable k is masked by a uniform random variable r in the *main* procedure, yielding two shares, i.e., r and $k_1 = k \oplus r$. Remark that this masking process should be performed outside of the program and the input of the *main* procedure is indeed the pair (r, k_1). We added here for verification purpose only. Then it invokes the procedure *SecExp254* to compute k^{254} using the shares (r, k_1).

The procedure *SecMult* is used to compute first-order secure multiplication over $\mathbb{GF}(2^8)$. Namely, given two shares (a_0, a_1) of a (i.e., $a_0 \oplus a_1 = a$) and two shares (b_0, b_1) of b (i.e., $b_0 \oplus b_1 = b$), it outputs two shares (x_5, x_7) such that $x_5 \oplus x_7 = a \odot b$. The procedure *RefreshMasks* is used to remask shares, which, given two shares (a_0, a_1) of a, outputs two shares (y_0, y_1) such that $y_0 \oplus y_1 = a_0 \oplus a_1 = a$. However, here y_0 and y_1 are masked by the new random variable r_1. The procedure *SecExp254* is used to compute the (first-order secure) exponentiation to the power 254 over $\mathbb{GF}(2^8)$. For two shares (a_0, a_1) of a, it outputs two shares (z_{16}, z_{17}) such that $z_{16} \oplus z_{17} = (a_0 \oplus a_1)^{254} = a^{254}$.

For the procedure *SecMult*, we have:

- $X^{SecMult} = \{r_0,\ x_0,\ \cdots,\ x_7\}$,
- $X_r^{SecMult} = \{r_0\}$,
- $X_a^{SecMult} = \{a_0,\ a_1,\ b_0,\ b_1\}$.

The partial computations of internal variables in $X^{SecMult}$ are listed below:

$$\begin{aligned}
&\mathcal{E}(r_0) = r_0,\\
&\mathcal{E}(x_0) = a_1 \odot b_0,\\
&\mathcal{E}(x_1) = a_0 \odot b_1,\\
&\mathcal{E}(x_2) = (a_0 \odot b_1) \oplus r_0,\\
&\mathcal{E}(x_3) = ((a_0 \odot b_1) \oplus r_0) \oplus (a_1 \odot b_0),\\
&\mathcal{E}(x_4) = a_0 \odot b_0,\\
&\mathcal{E}(x_5) = (a_0 \odot b_0) \oplus r_0,\\
&\mathcal{E}(x_6) = a_1 \odot b_1,\\
&\mathcal{E}(x_7) = (a_1 \odot b_1) \oplus (((a_0 \odot b_1) \oplus r_0) \oplus (a_1 \odot b_0)).
\end{aligned}$$

For the procedure *SecExp254*, we have:

- $X^{SecExp254} = \{z_0,\ \cdots,\ z_{17}\}$,
- $X_r^{SecExp254} = \emptyset$,
- $X_a^{SecExp254} = \{a_0,\ a_1\}$.

By inlining the procedure call *RefreshMasks*(z_0, z_1) at the call-site 24, we obtain the procedure $\texttt{inline}(SecExp254, 24)$, as shown in Figure 4.

For the procedure $\texttt{inline}(SecExp254, 24)$, we have:

- $X^{\texttt{inline}(SecExp254,24)} = \{z_0,\ \cdots,\ z_{17},\ a_0@24,\ a_1@24,$ $r_1@24,\ y_0@24,\ y_1@24\}$,
- $X_r^{\texttt{inline}(SecExp254,24)} = \{r_1@24\}$,
- $X_a^{\texttt{inline}(SecExp254,24)} = \{a_0,\ a_1\}$.

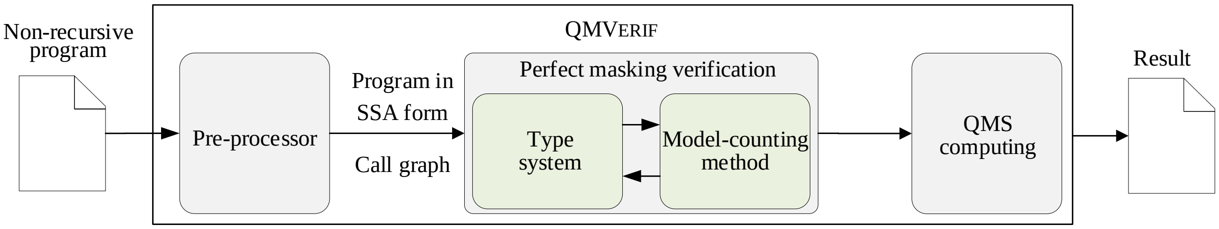


Fig. 3. Overview of our approach.

```
SecExp254($a_0, a_1$){                    //$a_0 \oplus a_1 = a$
   $z_0 = a_0^2$;
   $z_1 = a_1^2$;                         //$z_0 \oplus z_1 = a^2$
   $a_0@24 = z_0$;
   $a_1@24 = z_1$;
   $r_1@24 = \$$;
   $y_0@24 = a_0@24 \oplus r_1@24$;
   $y_1@24 = a_1@24 \oplus r_1@24$;
   $z_2 = y_0@24$;
   $z_3 = y_1@24$;                         //$z_2 \oplus z_3 = a^2$
   $z_4, z_5$ =SecMult($z_2, z_3, a_0, a_1$);   //$z_4 \oplus z_5 = a^3$
   $z_6 = z_4^4$;
   $z_7 = z_5^4$;                              //$z_6 \oplus z_7 = a^{12}$
   $z_8, z_9$ =RefreshMasks($z_6, z_7$);      //$z_8 \oplus z_9 = a^{12}$
   $z_{10}, z_{11}$ =SecMult($z_4, z_5, z_8, z_9$);  //$z_{10} \oplus z_{11} = a^{15}$
   $z_{12} = z_{10}^{16}$;
   $z_{13} = z_{11}^{16}$;                        //$z_{12} \oplus z_{13} = a^{240}$
   $z_{14}, z_{15}$ =SecMult($z_{12}, z_{13}, z_8, z_9$); //$z_{14} \oplus z_{15} = a^{252}$
   $z_{16}, z_{17}$ =SecMult($z_{14}, z_{15}, z_2, z_3$); //$z_{16} \oplus z_{17} = a^{254}$
   return  $z_{16}, z_{17}$;
}
```

Fig. 4. The procedure inline($SecExp254, 24$).

3.2 Approach Overview

An overview of our approach is given in Figure 3, which consists of four components: pre-processor, type system, model-counting method and QMS computing.

For a given non-recursive program P, the pre-processor transforms P to an equivalent program P' in the SSA form and constructs the call graph of P'. At a high-level, the type system is used to quickly obtain soundness proofs when an internal variable is perfectly masked. To resolve instances that cannot be inferred by the type system, the model-counting method is applied which is, in theory, powerful enough to completely determine if an internal variable is perfectly masked or leaky. Regardless of whether it is perfectly masked, the result is fed back to improve the type inference. Finally, based on the refined type inference result, we continue to analyze other internal variables. For the leaky variables, the QMS computing component can be applied to compute their QMS values by leveraging the model-counting method.

To verify the program P' using the type system, one possible way is to transform it into the equivalent program $P'_{\texttt{inlined}}$ by inlining all the procedures and then verify all the full computations of $P'_{\texttt{inlined}}$. The shortcoming of this approach is that some variables in P' may need to be analyzed multiple times (e.g., variables in *RefreshMasks* and *SecMult* in Figure 2), which may disadvantage scalability. We hence propose a *compositional* verification technique to address this issue. We directly analyze the program P' instead of $P'_{\texttt{inlined}}$. The subprocedures within P' which are invoked multiple times are to be analyzed in isolation in the reverse topological order of its call graph (note that P' is non-recursive, so the call graph is essentially a DAG), and the results are composed to give the overall verification.

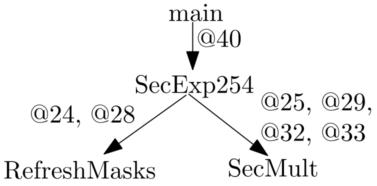


Fig. 5. Call graph of the program P^{254}.

For instance, the call graph of the program P^{254} is shown in Figure 5, where the labels on edges denote call-sites. We can verify the procedures in the order of (*RefreshMasks*, *SecMult*, *SecExp254*, *main*) or (*SecMult*, *RefreshMasks*, *SecExp254*, *main*).

To verify each procedure in isolation, the main challenge is to verify partial computations, that may be dependent on external variables of the procedure. For instance, consider an internal variable x_0 that is defined in *SecMult*, which corresponds to the variables $\{x_0@25@40, x_0@29@40, x_0@32@40, x_0@33@40\}$, i.e., $\texttt{inline}(x)$ in $P^{254}_{\texttt{inlined}}$. It is impossible to obtain the full computations $\mathcal{E}(x_0@25@40)$, $\mathcal{E}(x_0@29@40)$, $\mathcal{E}(x_0@32@40)$ and $\mathcal{E}(x_0@33@40)$ in isolation, as they rely upon the formal arguments a_1 and b_0. Likewise, the variable z_2 in *SecExp254* corresponds to the variable $z_2@40$ in the program $P^{254}_{\texttt{inlined}}$, which relies upon the first return value of the procedure call *RefreshMasks*(z_0, z_1), i.e., *RefreshMasks*$(z_0, z_1)[1]$. To address this challenge, we adopt the assume-guarantee reasoning [59], which is a modular technique that uses assumptions when checking procedures in isolation.

In our assume-guarantee framework, each procedure $f(a_1, \cdots, a_m)$ can be annotated with an assumption Φ_f by the user which expresses the properties of the formal arguments $a_1, \cdots, a_m$. For this purpose, we introduce a simple logic. For each internal variable $x \in X^f$, we infer the distribution type of the partial computation $\mathcal{E}(x)$ under the annotated assumption Φ_f via our type system. If the partial computation $\mathcal{E}(x)$ is statistically independent of the private input variables under the annotated assumption Φ_f and the actual arguments of the procedure call $f(x_1, \cdots, x_m)$

at the call-site ℓ do satisfy the assumption, we can deduce that the partial computation $\mathcal{E}(x@\ell)$ is indeed statistically independent of the private input variables after inlining this procedure. Finally, if all the procedure calls from the *main* procedure to the procedure $f(x_1, \cdots, x_m)$ with the sequence of the call-sites $\ell_1, \cdots, \ell_k, \ell$ satisfy the corresponding assumptions, then we can deduce that the full computation $\mathcal{E}(x@\ell@\ell_k \cdots @\ell_1)$ in the program $P_{\texttt{inlined}}$ is indeed perfectly masked. By this way, the verification obligation is to check whether each procedure in isolation is perfectly masked under the annotated assumption, and that the assumption of each procedure call holds. Note there is no guarantee that this approach can always work successfully. In case that the type inference fails on $\mathcal{E}(x)$ or the procedure call does not satisfy the corresponding assumption, we will inline the procedure call and apply the type system on the partial computation $\mathcal{E}(x@\ell)$. This procedure is repeated until it is proved or becomes the full computation. If the type system still fails on the full computation, we resort to (expensive) model-counting which is powerful enough to completely decide if the full computation is leakage-free.

4 Methodology

In this section, we present our type system, model-counting method, domain specific heuristics and overall algorithms.

4.1 Type System

We first introduce the distribution types, the notion of dominant variables which will be used in the type inference rules, and a simple logic for expressing the assumptions of procedures. We then present the type inference rules and explain how to deal with procedure calls.

4.1.1 Distribution Types

In our type system, there are four distribution types: τ_{uf}, τ_{si}, τ_{lk} and τ_{uk}. We denote by $\mathcal{T}$ the set $\{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}, \tau_{\mathsf{lk}}, \tau_{\mathsf{uk}}\}$. Intuitively, for each (partial or full) computation e,

- $e : \tau_{\mathsf{uf}}$ meaning that the distribution of the values of e is uniform;
- $e : \tau_{\mathsf{si}}$ meaning that the distribution of the values of e is statistically independent on private inputs;
- $e : \tau_{\mathsf{lk}}$ meaning that the distribution of the values of e is statistically dependent on private inputs;
- $e : \tau_{\mathsf{uk}}$ meaning that the distribution of the values of e is unknown.

where τ_{uf} is a subtype of τ_{si}.

Given a procedure $f(a_1, \cdots, a_m)$, let $\Phi_f = \{\psi_1, \cdots, \psi_k\}$ be the annotated assumption of f (the language for expressing formulas ψ_i will be defined in Section 4.1.3). To infer the distribution type of a partial computation e, the type judgement of e is defined in the form of

$$\Phi_f \vdash e : \tau,$$

where $\tau \in \mathcal{T}$ denotes the distribution type of the partial computation e under the assumption Φ_f.

A type judgement $\Phi_f \vdash e : \tau$ is valid if the type judgement $\psi \vdash e : \tau$ is valid for every formula $\psi \in \Phi_f$. We will present type inference rules in Section 4.1.4 to derive the valid type judgement $\psi \vdash e : \tau$. We say the procedure $f(a_1, \cdots, a_m)$ is *perfectly masked* under the assumption Φ_f, if for every variable $x \in X^f$, either $\Phi_f \vdash \mathcal{E}(x)_{\texttt{inlined}} : \tau_{\mathsf{uf}}$ or $\Phi_f \vdash \mathcal{E}(x)_{\texttt{inlined}} : \tau_{\mathsf{si}}$ is valid.

4.1.2 Dominant Variables

Given a computation e, a random variable r is a *dominant variable* of e if the following conditions hold:

1) r (syntactically) occurs in e exactly once,
2) and in the abstract syntax tree of e, for each operator $\circ \in \mathcal{O}^*$ on the path between the root and the leaf labeled by r, one of the following cases holds:
   - $\circ = \odot$ and one of its children is a non-zero constant;
   - $\circ \in \{\oplus, \neg, +, -\}$;
   - $\circ$ is a (univariate) bijective function, e.g., Sbox.

For efficiency consideration, to determine whether a random variable is dominant or not, we take a purely syntactic approach. For instance, r is *not* considered to be a dominant variable in $r \oplus ((r \oplus y) \oplus r)$, although r is a dominant variable in the equivalent $y \oplus r$. We will address this limitation in Section 4.3.

Intuitively, for every variable $x \in X^f$, if r is a dominant variable of the computation $\mathcal{E}(x)_{\texttt{inlined}}$, then the distribution of $\mathcal{E}(x)_{\texttt{inlined}}$ is uniform, as the random variables in $\mathcal{E}(x)_{\texttt{inlined}}$ are uniformly distributed.

Let $\mathsf{Var}(e)$ denote the set of variables used in the computation e, and $\mathsf{RVar}(e) \subseteq \mathsf{Var}(e)$ the set of random variables. We denote by $\mathsf{Dom}(e) \subseteq \mathsf{RVar}(e)$ the set of all dominant random variables of e. All these sets can be computed in linear time in the size of e. It is straightforward to have:

Proposition 1. If $\mathsf{Dom}(\mathcal{E}(x)_{\texttt{inlined}}) \neq \emptyset$, then the distribution of the values of $\mathcal{E}(x)_{\texttt{inlined}}(\sigma)$ is uniform for all possible assignments σ of formal arguments and variables in $\mathcal{E}(x)_{\texttt{inlined}}$.

4.1.3 A Logic for Expressing Assumptions

To express assumptions of procedures, we introduce a simple logic to specify properties of formal arguments. For each procedure $f(a_1, \cdots, a_m)$, its assumption is given as a set of formulas $\Phi_f = \{\psi_1, \cdots, \psi_k\}$, such that every $1 \leq i \leq k$, the formula ψ_i is defined by the following logic:

$$\begin{aligned}\phi ::= \top \mid a_i : \tau \mid \mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset \\ \mid \phi_1 \wedge \phi_2 \mid \mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset\end{aligned}$$

where $\tau \in \mathcal{T}$ is a distribution type, a_i and a_j are formal arguments of $f(a_1, \cdots, a_m)$.

A procedure call $f(x_1, \cdots, x_m)$ made in the procedure g satisfies the assumption Φ_f, denoted by $f(x_1, ..., x_m) \models \Phi_f$, iff $f(x_1, \cdots, x_m) \models \psi$ for some $\psi \in \Phi_f$, where the latter is inductively defined as follows:

- $f(x_1, \cdots, x_m) \models \top$ always holds;
- $f(x_1, \cdots, x_m) \models a_i : \tau$ iff $\Phi_g \vdash \mathcal{E}(a_i) : \tau$ is valid, where Φ_g denotes the assumption of the procedure g;
- $f(x_1, \cdots, x_m) \models \mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset$ iff $\mathsf{Dom}(\mathcal{E}(x_i)_{\texttt{inlined}}) \setminus \mathsf{RVar}(\mathcal{E}(x_j)_{\texttt{inlined}}) \neq \emptyset$;
- $f(x_1, \cdots, x_m) \models \mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset$ iff $\mathsf{RVar}(\mathcal{E}(x_i)_{\texttt{inlined}}) \cap \mathsf{RVar}(\mathcal{E}(x_j)_{\texttt{inlined}}) = \emptyset$
- $f(x_1, \cdots, x_m) \models \phi_1 \wedge \phi_2$ iff both $f(x_1, \cdots, x_m) \models \phi_1$ and $f(x_1, \cdots, x_m) \models \phi_2$.

$$\frac{\mathsf{Dom}(e) \neq \emptyset}{\psi \vdash e : \tau_{\mathsf{uf}}}\ (\textsc{Dom}) \qquad \frac{\psi \vdash e_1 \star e_2 : \tau}{\psi \vdash e_2 \star e_1 : \tau}\ (\textsc{Com}) \qquad \frac{\begin{array}{c} e \text{ is a full computation} \\ \mathsf{Var}(e) \cap X_k = \emptyset \end{array}}{\psi \vdash e : \tau_{\mathsf{si}}}\ (\textsc{NoKey})$$

$$\frac{x \in X_k}{\psi \vdash x : \tau_{\mathsf{lk}}}\ (\textsc{Key}) \qquad \frac{\psi \vdash e : \tau}{\psi \vdash \neg e : \tau}\ (\textsc{Ide}_1) \qquad \frac{\psi \vdash e : \tau_{\mathsf{si}}}{\psi \vdash e \bullet e : \tau_{\mathsf{si}}}\ (\textsc{Ide}_2)$$

$$\frac{}{\psi \vdash e \diamond e : \tau_{\mathsf{si}}}\ (\textsc{Ide}_3) \qquad \frac{\psi \vdash e : \tau_{\mathsf{lk}}}{\psi \vdash e \bowtie e : \tau_{\mathsf{lk}}}\ (\textsc{Ide}_4) \qquad \frac{\begin{array}{c} \psi \vdash e_1 : \tau_{\mathsf{uf}} \quad \psi \vdash e_2 : \tau_{\mathsf{si}} \\ \psi \vdash \mathsf{Dom}(e_1) \setminus \mathsf{RVar}(e_2) \neq \emptyset \end{array}}{\psi \vdash e_1 \circ e_2 : \tau_{\mathsf{si}}}\ (\textsc{Sid}_1)$$

$$\frac{\begin{array}{c} \psi \vdash e_1 : \tau_{\mathsf{si}} \quad \psi \vdash e_2 : \tau_{\mathsf{si}} \\ \psi \vdash \mathsf{RVar}(e_1) \cap \mathsf{RVar}(e_2) = \emptyset \end{array}}{\psi \vdash e_1 \bullet e_2 : \tau_{\mathsf{si}}}\ (\textsc{Sid}_2) \qquad \frac{\begin{array}{c} \psi \vdash e_1 : \tau_{\mathsf{lk}} \quad \psi \vdash e_2 : \tau_{\mathsf{uf}} \\ \psi \vdash \mathsf{Dom}(e_2) \setminus \mathsf{RVar}(e_1) \neq \emptyset \end{array}}{\psi \vdash e_1 \circ e_2 : \tau_{\mathsf{lk}}}\ (\textsc{Leak}) \qquad \frac{\begin{array}{c} \psi \vdash \texttt{inline}(e, \ell) : \tau \\ \ell \text{ is a call-site} \end{array}}{\psi \vdash e : \tau}\ (\textsc{Inline})$$

$$\frac{\text{No rule is appliable to } e}{\psi \vdash e : \tau_{\mathsf{uk}}}\ (\textsc{Ukd}) \qquad \frac{\mathsf{Dom}(e_1) \setminus \mathsf{RVar}(e_2) \neq \emptyset}{\psi \vdash \mathsf{Dom}(e_1) \setminus \mathsf{RVar}(e_2) \neq \emptyset} \qquad \frac{\mathsf{RVar}(e_{1\texttt{inlined}}) \cap \mathsf{RVar}(e_{2\texttt{inlined}}) = \emptyset}{\psi \vdash \mathsf{RVar}(e_1) \cap \mathsf{RVar}(e_2) = \emptyset}$$

$$\frac{a_i : \tau \text{ is a conjunct of } \psi}{\psi \vdash a_i : \tau}\ (\textsc{Apt}) \qquad \frac{\begin{array}{c} \mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset \\ \text{is a conjunct of } \psi \end{array}}{\psi \vdash \mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset} \qquad \frac{\begin{array}{c} \mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset \\ \text{is a conjunct of } \psi \end{array}}{\psi \vdash \mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset}$$

Fig. 6. Type inference rules, where $\star \in \mathcal{O}$, $\circ \in \{\wedge, \vee, \odot, \times\}$, $\bullet \in \mathcal{O}^*$, $\bowtie \in \{\wedge, \vee\}$ and $\diamond \in \{\oplus, -\}$, a_i and a_j denote formal arguments of the procedure $f(a_1, \cdots, a_m)$.

Given a sequence π of procedure calls $f_1(x_1^1, \cdots, x_{m_1}^1)$, $\cdots, f_k(x_1^k, \cdots, x_{m_k}^k)$, let $\pi_\ell = \ell_1, \cdots, \ell_k$ denote the sequence of the corresponding call-sites, and Φ_π denote the sequence of assumptions $\Phi_{f_1}, \cdots, \Phi_{f_k}$. We say π satisfies Φ_π, denoted by $\pi \models \Phi_\pi$, if $f_i(x_1^i, \cdots, x_{m_i}^i) \models \Phi_{f_i}$ for every $1 \leq i \leq k$.

We remark that the assumption of the *main* procedure is not needed, as there is no procedure call to the *main* procedure. (Alternatively one can assume it to be $\{\top\}$.)

4.1.4 Type Inference Rules

Given a procedure $f(a_1, \cdots, a_m)$, for every variable $x \in X^f$ such that $\mathcal{E}(x) = e$, to derive valid type judgements $\psi \vdash e : \tau$ for all formulas $\psi \in \Phi_f$, we design type inference rules (shown in Figure 6). We will drop the context ψ from $\psi \vdash e : \tau$ when $\psi = \top$.

Rule (Dom) directly follows Proposition 1. Rule (Com) follows the commutative law of operations $\star \in \mathcal{O}$. Rule (NoKey) describes that full computations without using any private input variables have type τ_{si}. Note that we cannot apply this rule to computations that contain some formal arguments, but are neither public nor private. Rule (Key) enures that each private input has type τ_{lk}. Rules (Ide$_i$) for $i = 1, 2, 3, 4$ are straightforward. Rule (Apt) follows from the assumption ψ.

Rule (Sid$_1$) states that if e_1 has type τ_{uf}, e_2 has type τ_{si}, and e_1 has a dominant variable r which is not used by e_2 (implying that $e_{2\texttt{inlined}}$ does not use r), then $e_1 \circ e_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$ has type τ_{si}. This is because that $e_1 \circ e_2$ can be seen as $r \circ e_2$, and the distributions of the values of r and e_2 are independent. In this rule, when e_1 and e_2 are formal arguments, we check the premise $\mathsf{Dom}(e_1) \setminus \mathsf{RVar}(e_2) \neq \emptyset$ using the type context ψ.

Likewise, if both e_1 and e_2 have type τ_{si} (as well as its subtype τ_{uf}), and e_1 and e_2 use disjoint random variables, then $e_1 \bullet e_2$ for $\bullet \in \mathcal{O}^*$ has type τ_{si}, as the distributions of values of e_1 and e_2 are independent. This is captured by rule (Sid$_2$). Similar to rule (Sid$_1$), rule (Leak) states that if e_1 has type τ_{lk}, e_2 has type τ_{uf}, and e_2 has a dominant variable r which is not used by e_1, we can deduce that $e_1 \circ e_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$ has type τ_{lk}.

Our type system is designed to infer types of partial computations for each procedure in isolation. Therefore, $\mathsf{Dom}(e)$ and $\mathsf{RVar}(e)$ rather than $\mathsf{Dom}(e_{\texttt{inlined}})$ and $\mathsf{RVar}(e_{\texttt{inlined}})$ are used in the type inference rules, according to Proposition 2. When it fails to derive a valid type judgement and e contains a procedure call with the call-site ℓ, rule (Inline) can be used to infer the type of e by inferring the type of $\texttt{inline}(e, \ell)$. These features allow to inline procedures as less as possible.

The type judgements derived by the above rules are conclusive, therefore the type system is sound. We also demonstrate in our experiments that for cryptographic programs, these rules suffice to drive type judgements of most computations. However, there may exist computations whose types cannot be inferred by the above rules. Therefore, for these computations, we design a specific rule (Ukd) which assigns unknown distribution type to these computations. We will address this problem for the full computations in Section 4.2 by leveraging model-counting methods. It is easy to see that:

Theorem 1. Given a program P, for every variable x defined in the program $P_{\texttt{inlined}}$,

- $P_{\texttt{inlined}}$ is x-**UF**, if $\vdash \mathcal{E}(x) : \tau_{\mathsf{uf}}$ is valid;
- $P_{\texttt{inlined}}$ is x-**SI**, if $\vdash \mathcal{E}(x) : \tau_{\mathsf{si}}$ is valid;
- $P_{\texttt{inlined}}$ is not x-**SI**, if $\vdash \mathcal{E}(x) : \tau_{\mathsf{lk}}$ is valid.

4.1.5 Compositional Property

To verify the program P but avoid a full construction of the program $P_{\texttt{inlined}}$, we show the following compositional property.

Theorem 2. Given a procedure $f(a_1, \cdots, a_m)$ of P, for every $x \in X^f$ and sequence π of procedure calls starting from one in the *main* procedure to f with $\pi_\ell = \ell_1 \cdots \ell_k$, if $\pi \models \Phi_\pi$ and $\Phi_f \vdash \mathcal{E}(x) : \tau$ is valid for $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}, \tau_{\mathsf{lk}}\}$, then $\vdash \mathcal{E}(x@\ell_k \cdots @\ell_1)_{\texttt{inlined}} : \tau$ is valid. (Note that $\mathcal{E}(x@\ell_k \cdots @\ell_1)_{\texttt{inlined}}$ is the full computation of the variable $x@\ell_k \cdots @\ell_1$ in $P_{\texttt{inlined}}$.)

By Theorems 1 and 2, we can deduce that the program $P_{\texttt{inlined}}$ is $x@\ell_k \cdots @\ell_1$-**UF** (resp. $x@\ell_k \cdots @\ell_1$-**SI** or not $x@\ell_k \cdots @\ell_1$-**SI**), if $\pi \models \Phi_\pi$ and $\Phi_f \vdash \mathcal{E}(x) : \tau_{\mathsf{uf}}$ (resp. $\Phi_f \vdash \mathcal{E}(x) : \tau_{\mathsf{si}}$ or $\Phi_f \vdash \mathcal{E}(x) : \tau_{\mathsf{lk}}$) is valid. The correctness of Theorem 2 directly follows from the following two lemmas.

The first lemma shows that it suffices to infer the distribution type of the observable variable x defined in a procedure from its partial computation $\mathcal{E}(x)$ using our type system.

Lemma 1. Given a procedure $f(a_1, \cdots, a_m)$, for every variable $x \in X^f$, formula $\psi \in \Phi_f$ and $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}, \tau_{\mathsf{lk}}\}$, $\psi \vdash \mathcal{E}(x) : \tau$ is valid iff $\psi \vdash \mathcal{E}(x)_{\texttt{inlined}} : \tau$ is valid.

The second lemma shows that the distribution type of the partial computation $\mathcal{E}(x@\ell_k \cdots @\ell_1)$ can be deduced from the distribution type of the partial computation $\mathcal{E}(x)$ when the corresponding procedure assumptions are satisfied by their procedure calls.

Lemma 2. For every path $\pi = f_1(x_1^1, \cdots, x_{m_1}^1), \cdots, f_k(x_1^k, \cdots, x_{m_k}^k)$ in the call graph of the program P with $\pi_\ell = \ell_1, \cdots, \ell_k$, suppose $f_1(x_1^1, \cdots, x_{m_1}^1)$ is made in the procedure g. For every $x \in X^{f_k}$ and $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}, \tau_{\mathsf{lk}}\}$, if $\pi \models \Phi_\pi$ and $\Phi_{f_k} \vdash \mathcal{E}(x) : \tau$ is valid, then $\Phi_g \vdash \mathcal{E}(x@\ell_k \cdots @\ell_1) : \tau$ is valid, where $x@\ell_k \cdots @\ell_1$ is the variable defined in the procedure $\texttt{inline}(g, \ell_1, \cdots, \ell_k)$.

Formal proofs of Lemma 1 and Lemma 2 are given in the supplemental material.

Remarkably, if $\vdash \mathcal{E}(x) : \tau$ is valid, we can still deduce $\Phi_g \vdash \mathcal{E}(x@\ell_k \cdots @\ell_1) : \tau$ even if $f_k(x_1^k, \cdots, x_{m_1}^k) \not\models \Phi_{f_k}$.

Example 1. Let us consider the program in Figure 2. Suppose the procedure *SecMult* is annotated with the assumption $\Phi_{SecMult} = \{\psi_1, \psi_2\}$, where

- $\psi_1 = \bigwedge_{0 \le i,j \le 1} (\mathsf{Dom}(a_i) \setminus \mathsf{RVar}(b_j) \neq \emptyset \wedge b_j : \tau_{\mathsf{si}})$,
- $\psi_2 = \bigwedge_{0 \le i,j \le 1} (\mathsf{Dom}(b_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset \wedge a_j : \tau_{\mathsf{si}})$.

The partial computations of variables $x \in X^{SecMult}$ are given in Section 3.1. For every $i \in \{1, 2\}$, by applying rule (SID$_1$), it is easy to derive:

$$\psi_i \vdash \mathcal{E}(x_0) : \tau_{\mathsf{si}}, \quad \psi_i \vdash \mathcal{E}(x_1) : \tau_{\mathsf{si}},$$
$$\psi_i \vdash \mathcal{E}(x_4) : \tau_{\mathsf{si}}, \quad \psi_i \vdash \mathcal{E}(x_6) : \tau_{\mathsf{si}}.$$

We can also deduce that $\mathcal{E}(r_0)$, $\mathcal{E}(x_2)$, $\mathcal{E}(x_3)$, $\mathcal{E}(x_5)$ and $\mathcal{E}(x_7)$ have type τ_{uf} by applying rule (DOM) even if the assumption $\Phi_{SecMult} = \{\top\}$, as they are dominated by a local random variable r_0 which never occur in $\mathcal{E}(a_0)$, $\mathcal{E}(a_1)$, $\mathcal{E}(b_0)$ and $\mathcal{E}(b_1)$.

Similarly, for the variables r_1, y_0 and y_1 defined in *RefreshMasks*, our type system can derive $\vdash \mathcal{E}(r_1) : \tau_{\mathsf{uf}}$, $\vdash \mathcal{E}(y_0) : \tau_{\mathsf{uf}}$ and $\vdash \mathcal{E}(y_1) : \tau_{\mathsf{uf}}$, as $\mathcal{E}(r_1)$, $\mathcal{E}(y_0)$ and $\mathcal{E}(y_1)$ are dominated by the random variable r_1.

For the procedure *SecExp254*, let $\Phi_{SecExp254} = \{a_0 : \tau_{\mathsf{uf}} \wedge a_1 : \tau_{\mathsf{uf}}\}$, we can derive

- $\Phi_{SecExp254} \vdash \mathcal{E}(z_0) : \tau_{\mathsf{si}}$ and $\Phi_{SecExp254} \vdash \mathcal{E}(z_1) : \tau_{\mathsf{si}}$ by applying rule (IDE$_2$),
- $\vdash \mathcal{E}(z_2) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_3) : \tau_{\mathsf{uf}} \vdash \mathcal{E}(z_4) : \tau_{\mathsf{uf}}$ and $\vdash \mathcal{E}(z_5) : \tau_{\mathsf{uf}}$ by applying rule (INLINE),
- $\vdash \mathcal{E}(z_6) : \tau_{\mathsf{si}}$ and $\vdash \mathcal{E}(z_7) : \tau_{\mathsf{si}}$ by applying rule (IDE$_2$),
- $\vdash \mathcal{E}(z_8) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_9) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_{10}) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_{11}) : \tau_{\mathsf{uf}}$ by applying rule (INLINE),
- $\vdash \mathcal{E}(z_{12}) : \tau_{\mathsf{si}}$ and $\vdash \mathcal{E}(z_{13}) : \tau_{\mathsf{si}}$ by applying rule (IDE$_2$),
- $\vdash \mathcal{E}(z_{14}) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_{15}) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_{16}) : \tau_{\mathsf{uf}}, \vdash \mathcal{E}(z_{17}) : \tau_{\mathsf{uf}}$ by applying rule (INLINE).

One can observe that the procedure call *SecExp254*(r, k_1) satisfies $a_0 : \tau_{\mathsf{uf}} \wedge a_1 : \tau_{\mathsf{uf}}$. Suppose $\Phi_{RefreshMasks} = \{\top\}$. It is easy to verify that all the procedure calls *SecMult*(z_2, z_3, a_0, a_1), *SecMult*(z_4, z_5, z_8, z_9), *SecMult*$(z_{12}, z_{13}, z_8, z_9)$ and *SecMult*$(z_{14}, z_{15}, z_2, z_3)$ satisfy either ψ_1 or ψ_2. Therefore, we can deduce that the program P^{254} is perfectly masked.

4.1.6 Reducing Procedure Inlines Further

One may observe that

1) to verify variables whose computations depend upon the return values of some procedure calls, we may have to apply the rule (INLINE) (e.g., z_2 and z_3 in Example 1);
2) to check whether a procedure call $f(y_1, \cdots, y_m)$ satisfies the formula $\mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset$ (resp. $\mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset$), we have to verify whether $\mathsf{Dom}(\mathcal{E}(y_i)_{\texttt{inlined}}) \setminus \mathsf{RVar}(\mathcal{E}(y_j)_{\texttt{inlined}}) \neq \emptyset$ (resp. $\mathsf{RVar}(\mathcal{E}(y_i)_{\texttt{inlined}}) \cap \mathsf{RVar}(\mathcal{E}(y_j)_{\texttt{inlined}}) = \emptyset$);
3) $\mathsf{RVar}(e_{1\,\texttt{inlined}}) \cap \mathsf{RVar}(e_{2\,\texttt{inlined}}) = \emptyset$ is a premise of a type inference rule in our type system (cf. Table 6).

Verifying these conditions requires procedure inlines. In this section, we present our solutions so that some procedure inlines can be avoided.

To tackle the first issue, consider the following function

$$f(a_1, \cdots, a_m) = s_1; \cdots s_t; \texttt{return } z_1, \cdots, z_k;.$$

For each variable x_i which is defined by $x_i = f(y_1, \cdots, y_m)[i]@\ell$; for some $1 \le i \le k$ after procedure inlining, we regard the partial computation $\mathcal{E}(x_i)$ (without inlining) as a special computation such that

- $\mathsf{Dom}(\mathcal{E}(x_i)) = \{r@\ell \mid r \in \mathsf{Dom}(\mathcal{E}(z_i))\}$;
- $\mathsf{RVar}(\mathcal{E}(x_i)) = \{r@\ell \mid r \in \mathsf{RVar}(\mathcal{E}(z_i))\} \cup RA$, where $RA = \bigcup_{a_i \in \mathsf{Var}(\mathcal{E}(z_i))} \mathsf{RVar}(\mathcal{E}(y_i))$ is the set of random variables used in the partial computations $\mathsf{RVar}(\mathcal{E}(y_i))$ of the actual parameters on which $\mathcal{E}(z_i)$ relies when the procedure call is inlined.

For each computation e that relies upon some return values x_i of the procedure call $f(y_1, \cdots, y_m)$, $\mathsf{RVar}(e)$ and $\mathsf{Dom}(e)$ can be extended accordingly by taking $\mathsf{RVar}(\mathcal{E}(x_i))$ and $\mathsf{Dom}(\mathcal{E}(x_i))$ into account for all return values x_i of the procedure call simultaneously. This is done only when the original sets $\mathsf{RVar}(e)$ and $\mathsf{Dom}(e)$ are insufficient. For instance, for each variable $z \in \{z_2, ..., z_5, z_8, ..., z_{11}, z_{14}, ..., z_{17}\}$ of the running example, we can deduce $\vdash \mathcal{E}(z) : \tau_{\mathsf{uf}}$ without applying the rule (INLINE), as $\mathsf{Dom}(\mathcal{E}(z)) \neq \emptyset$.

To tackle the second issue, we first consider the formula $\mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset$. For every variable $x \in X^f$ and procedure call at the call-site ℓ in the procedure $f(a_1, \cdots, a_m)$, it is easy to see that $\mathsf{Dom}(\mathcal{E}(x)) \subseteq \mathsf{Dom}(\texttt{inline}(\mathcal{E}(x), \ell))$. Indeed, if there exists some $r \in \mathsf{Dom}(\mathcal{E}(x))$, then r must be a local variable of f, implying that r is used in

- neither $\mathcal{E}(x_1)_{\texttt{inlined}}, \cdots, \mathcal{E}(x_m)_{\texttt{inlined}}$ for all procedure calls $f(x_1, \cdots, x_m)$,
- nor the procedure call at the call-site ℓ.

Therefore, $r \in \mathsf{Dom}(\texttt{inline}(\mathcal{E}(x), \ell))$. Similarly, for every pair of variables $x, x' \in X^f$, we have that

$$\mathsf{Dom}(\mathcal{E}(x')) \setminus \mathsf{RVar}(\mathcal{E}(x)) = \mathsf{Dom}(\mathcal{E}(x')) \setminus \mathsf{RVar}(\texttt{inline}(\mathcal{E}(x), \ell)).$$

By leveraging the solution to the first issue, we have that, if $\mathsf{Dom}(\mathcal{E}(y_i)) \setminus \mathsf{RVar}(\mathcal{E}(y_j)) \neq \emptyset$, then $\mathsf{Dom}(\mathcal{E}(y_i)_{\texttt{inlined}}) \setminus \mathsf{RVar}(\mathcal{E}(y_j)_{\texttt{inlined}}) \neq \emptyset$. This often allows us to prove that the procedure call $f(y_1, \cdots, y_m)$ satisfies the formula $\mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset$ without fully inlining all the procedure calls.

Example 2. Let us consider the procedure *SecExp254* in the running example. Since $\mathsf{Dom}(\mathcal{E}(y_0)) = \{r_1\}$, $\mathsf{Dom}(\mathcal{E}(z_2)) = \emptyset$ can be refined to the set $\mathsf{Dom}(\mathcal{E}(z_2)) = \{r_1@24\}$. From $\mathsf{RVar}(\mathcal{E}(a_0)) = \emptyset$, we can get that $\mathsf{Dom}(\mathcal{E}(z_2)_{\texttt{inlined}}) \setminus \mathsf{RVar}(\mathcal{E}(a_0)_{\texttt{inlined}}) \neq \emptyset$, hence *SecMult*$(z_2, z_3, a_0, a_1)$ satisfies the formula $\mathsf{Dom}(a_0) \setminus \mathsf{RVar}(b_0) \neq \emptyset$, which is a conjunct of the annotation ψ_1 of the procedure *SecMult*.

Note that currently we cannot prove that *SecMult*$(z_{12}, z_{13}, z_8, z_9)$ satisfies $\mathsf{Dom}(b_0) \setminus \mathsf{RVar}(a_0) \neq \emptyset$, as the random variable $r_1@28 \in \mathsf{Dom}(\mathcal{E}(z_8))$ occurs in $\mathsf{RVar}(\mathcal{E}(z_{10}))$, hence also occurring in $\mathsf{RVar}(\mathcal{E}(z_{12}))$. We will address this problem in Section 4.3.

We consider $\mathsf{RVar}(e_{1\texttt{inlined}}) \cap \mathsf{RVar}(e_{2\texttt{inlined}}) = \emptyset$ which is involved in both the second and the third issue. This is much more involved, as $\mathsf{RVar}(\mathcal{E}(x)) \cap \mathsf{RVar}(\mathcal{E}(x')) = \emptyset$ does not imply $\mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x), \ell)) \cap \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x'), \ell)) = \emptyset$ when the inlined procedure at the call-site ℓ introduces random variables that occur in both $\texttt{inlined}(\mathcal{E}(x), \ell)$ and $\texttt{inlined}(\mathcal{E}(x'), \ell)$. This means that even when $\mathsf{RVar}(\mathcal{E}(a_i)) \cap \mathsf{RVar}(\mathcal{E}(a_j)) = \emptyset$, $\mathsf{RVar}(\mathcal{E}(a_i)_{\texttt{inlined}}) \cap \mathsf{RVar}(\mathcal{E}(a_j)_{\texttt{inlined}}) = \emptyset$ may not hold.

To address this problem, we write $\mathcal{E}(x) \dagger \mathcal{E}(x')$ if $\mathcal{E}(x)$ and $\mathcal{E}(x')$ do *not* involve the same procedure call. If $\mathcal{E}(x) \dagger \mathcal{E}(x')$ and $\mathsf{RVar}(\mathcal{E}(x)) \cap \mathsf{RVar}(\mathcal{E}(x')) = \emptyset$ both hold, we have that

- $\mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x), \ell)) \cap \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x'), \ell)) = \emptyset$
- and $\texttt{inlined}(\mathcal{E}(x), \ell) \dagger \texttt{inlined}(\mathcal{E}(x'), \ell)$.

This implies that, if $\texttt{inlined}(\mathcal{E}(a_i), \ell) \dagger \texttt{inlined}(\mathcal{E}(a_j), \ell)$ and $\mathsf{RVar}(\texttt{inlined}(\mathcal{E}(a_i), \ell)) \cap \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(a_j), \ell)) = \emptyset$ for some call-site ℓ, then we have: $\mathsf{RVar}(\mathcal{E}(a_i)_{\texttt{inlined}}) \cap \mathsf{RVar}(\mathcal{E}(a_j)_{\texttt{inlined}}) = \emptyset$. The third issue can be handled similarly.

The above observations are summarized by the following proposition.

Proposition 2. For two variables $x, x' \in X^f$ of the procedure $f(a_1, \cdots, a_m)$ and a procedure call g at ℓ in $\mathcal{E}(x)$,

1) $\mathsf{Dom}(\mathcal{E}(x)) \subseteq \mathsf{Dom}(\texttt{inlined}(\mathcal{E}(x), \ell))$ $\subseteq \mathsf{Dom}(\mathcal{E}(x)_{\texttt{inlined}})$;
2) $\mathsf{Dom}(\mathcal{E}(x')) \setminus \mathsf{RVar}(\mathcal{E}(x))$ $= \mathsf{Dom}(\mathcal{E}(x')) \setminus \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x), \ell))$ $= \mathsf{Dom}(\mathcal{E}(x')) \setminus \mathsf{RVar}(\mathcal{E}(x)_{\texttt{inlined}})$ $\subseteq \mathsf{Dom}(\texttt{inlined}(\mathcal{E}(x'), \ell)) \setminus \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x), \ell))$;
3) if $\mathcal{E}(x) \dagger \mathcal{E}(x')$ and $\mathsf{RVar}(\mathcal{E}(x)) \cap \mathsf{RVar}(\mathcal{E}(x')) = \emptyset$, then $\mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x), \ell)) \cap \mathsf{RVar}(\texttt{inlined}(\mathcal{E}(x'), \ell)) = \emptyset$;
4) if $\mathsf{Dom}(\mathcal{E}(x)) \neq \emptyset$, then the distribution of the computation $\mathcal{E}(x)_{\texttt{inlined}}$ is uniform for all possible assignments σ of formal arguments and variables in $\mathcal{E}(x)_{\texttt{inlined}}$.

Algorithm 1: A brute-force algorithm

```
1  Function BFENUM($P, x, q$)
2      $m$ = number of bits of random variables in $\mathcal{E}(x)$;
3      $\Delta_x^q = (1-q) \times 2^m$;
4      forall $\eta_p : (X_p \cap \mathsf{Var}(\mathcal{E}(x))) \to \mathfrak{D}$ do
5          $D_1$ = a map with domain $\mathfrak{D}$ such that for all
             $c \in \mathfrak{D}.D_1(c) = 0$;
6          $b$ = false;
7          forall $\eta_k : (X_k \cap \mathsf{Var}(\mathcal{E}(x))) \to \mathfrak{D}$ do
8              $D_2$ = a map with domain $\mathfrak{D}$ such that for all
                 $c \in \mathfrak{D}.D_2(c) = 0$;
9              if $b$ == false then
10                 $D_1$ =COUNTING($P, x, \eta_p, \eta_k$);
11                 $b$ = true;
12             else
13                 $D_2$ =COUNTING($P, x, \eta_p, \eta_k$);
14                 if $\max_{c \in \mathfrak{D}} |D_1[c] - D_2[c]| > \Delta_x^q$ then
15                     return UNSAT
16     return SAT;

17 Function COUNTING($P, x, \eta_p, \eta_k$)
18     forall $\eta_r : \mathsf{RVar}(\mathcal{E}(x)) \to \mathfrak{D}$ do
19         $c$ = the value of $\mathcal{E}(x)$ under $\eta_p$, $\eta_k$ and $\eta_r$;
20         $D[c]$++;
21     return $D$;
```

4.2 Model-Counting based Methods

In this subsection, we propose two model-counting based methods for checking whether $\mathtt{QMS}_x \geq q$ for a given rational number $q \in [0, 1]$. Recalling that a program is x-**SI** iff $\mathtt{QMS}_x = 1$, hence, we can verify whether a program is x-**SI** by checking whether $\mathtt{QMS}_x \geq 1$. Indeed, the program is x-**SI** iff $\mathtt{QMS}_x \geq 1$ holds. On the other hand, we will present a binary search based algorithm for computing QMS values by iteratively querying $\mathtt{QMS}_x \geq q$ (cf. Section 4.4.2). Note that model-counting based methods are performed on full computations instead of partial computations.

4.2.1 Brute-force Method

Recall that $\mathtt{QMS}_x := 1 - \max_{(\sigma_1, \sigma_2) \in \Theta^2_{X_p}, c \in \mathfrak{D}} (\llbracket x \rrbracket_{\sigma_1}(c) - \llbracket x \rrbracket_{\sigma_2}(c))$. To check whether $\mathtt{QMS}_x \geq q$, the brute-force method (cf. Algorithm 1): (1) enumerates all possible assignments η_p of public input variables (Line 4), (2) for each η_p, enumerates all possible assignments η_k of private input variables (Line 7), and (3) for each pair of η_k and η'_k (function COUNTING), computes corresponding distributions $\llbracket x \rrbracket_{\eta_p, \eta_k}$ and $\llbracket x \rrbracket_{\eta_p, \eta'_k}$ again by enumerating the assignments η_r of random variables (Line 18). The distribution $\llbracket x \rrbracket_{\eta_p, \eta_k}$ (resp. $\llbracket x \rrbracket_{\eta_p, \eta'_k}$) is stored as an array D_1 (resp. D_2) in which each entity indexed by c is the number of assignments η_r of random variables such that the full computation $\mathcal{E}(x)$ evaluates to c under η_p, η_k and η_r (resp. η_p, η'_k and η_r). Once $\max_{c \in \mathfrak{D}} |D_1[c] - D_2[c]| > \Delta_x^q$ holds, we can deduce that $\mathtt{QMS}_x \geq q$ does not hold.

Theorem 3. Given a program P, for every variable x of $P_{\texttt{inlined}}$, $\mathtt{QMS}_x \geq q$ iff BFENUM$(P_{\texttt{inlined}}, x, q)$ returns SAT.

The complexity of Algorithm 1 is exponential in the number of (bits of) variables in $\mathcal{E}(x)$, so it would experience significant performance degradation when facing a large number of variables.

4.2.2 SMT-based Method

The SMT-based method is a generalization of the one proposed by Eldib et al. [49], [50] from the Boolean setting to the arithmetic one.

For a given variable x, a valuation $\sigma \in \Theta$ and a constant $c \in \mathfrak{D}$, we denote by $\sharp(c = [\![x]\!]_{\sigma})$ the number of assignments of random variables under which the full computation $\mathcal{E}(x)(\sigma)$ evaluates to c. Then, checking whether $\texttt{QMS}_x \geq q$ can be reduced to checking the unsatisfiability of the following model-counting constraint:

$$\exists c \in \mathfrak{D}.\exists \sigma_1, \sigma_2 \in \Theta^2_{X_p}.(\sharp(c = [\![x]\!]_{\sigma_1}) - \sharp(c = [\![x]\!]_{\sigma_2})) > \Delta^q_x \quad (1)$$

where $\Delta^q_x = (1-q) \times 2^m$, and m is the number of bits of random variables in $\mathcal{E}(x)$. Indeed,

$$\boxed{\begin{array}{c}
\texttt{QMS}_x \geq q \text{ holds} \\
\text{iff} \\
1 - \max\limits_{(\sigma_1,\sigma_2)\in\Theta^2_{X_p}, c\in\mathfrak{D}} ([\![x]\!]_{\sigma_1}(c) - [\![x]\!]_{\sigma_2}(c)) \geq q \text{ holds} \\
\text{iff} \\
\max\limits_{(\sigma_1,\sigma_2)\in\Theta^2_{X_p}, c\in\mathfrak{D}} ([\![x]\!]_{\sigma_1}(c) - [\![x]\!]_{\sigma_2}(c)) \leq 1 - q \text{ holds} \\
\text{iff} \\
\max\limits_{(\sigma_1,\sigma_2)\in\Theta^2_{X_p}, c\in\mathfrak{D}} \left(\sharp(c = [\![x]\!]_{\sigma_1}) - \sharp(c = [\![x]\!]_{\sigma_2})\right) \leq \Delta^q_x \text{ holds} \\
\text{iff} \\
\text{Eqn. (1) does not hold.}
\end{array}}$$

Furthermore, Eqn. (1) can be encoded as a (quantifier-free) first-order formula Ψ^q_x to be solved by an off-the-shelf SMT solver such as Z3 [47]:

$$\Psi^q_x := \left(\bigwedge_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathfrak{D}} (\Theta_f \wedge \Theta'_f) \right) \wedge \Theta_{\texttt{b2i}} \wedge \Theta'_{\texttt{b2i}} \wedge \Theta^q_{\texttt{diff}}$$

where

- **Program logic** (Θ_f and Θ'_f). For every assignment of random variables $f : \mathsf{RVar}(\mathcal{E}(x)) \to \mathfrak{D}$, the logical formula Θ_f encodes the computation $\mathcal{E}(x)$ in the way that each occurrence of a random variable $r \in \mathsf{RVar}(\mathcal{E}(x))$ is instantiated by its concrete value $f(r)$ and asserts that the value of $\mathcal{E}(x)$ equals to a fresh variable c_f.

  Θ'_f is similar to Θ_f with the exception that c_f and variables $k \in X_k$ are replaced by fresh variables c'_f and k' respectively.

  Note that there are $|\mathfrak{D}|^{|\mathsf{RVar}(\mathcal{E}(x))|}$ distinct copies of Θ_f (resp. Θ'_f) that share the same variables from X_p and X_k.

- **Boolean to integer** ($\Theta_{\texttt{b2i}}$ and $\Theta'_{\texttt{b2i}}$). The logical formula $\Theta_{\texttt{b2i}}$ asserts that for each assignment of random variables $f : \mathsf{RVar}(\mathcal{E}(x)) \to \mathfrak{D}$, a fresh integer variable I_f is 1 if $c = c_f$, and 0 otherwise. In this way, we can count the number of assignments of random variables under which $\mathcal{E}(x)$ evaluates to c by accumulating I_f's. Formally,

  $$\Theta_{\texttt{b2i}} := \bigwedge_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathfrak{D}} I_f = (c = c_f) \ ?\ 1 : 0.$$

  $\Theta'_{\texttt{b2i}}$ is similar to $\Theta_{\texttt{b2i}}$ except that I_f and c_f are replaced by I'_f and c'_f respectively.

- **Different sums** ($\Theta^q_{\texttt{diff}}$). $\Theta^q_{\texttt{diff}}$ asserts that the difference between the number of assignments of random variables under which the computations $\mathcal{E}(x)$ and $\mathcal{E}(x)'$ evaluate to c is greater than Δ^q_x, where $\mathcal{E}(x)'$ denotes the computation $\mathcal{E}(x)$ in which the private variables k are replaced by k'. Formally,

  $$\Theta^q_{\texttt{diff}} := \sum_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathfrak{D}} I_f - \sum_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathfrak{D}} I'_f > \Delta^q_x.$$

Theorem 4. $\texttt{QMS}_x \geq q$ iff Ψ^q_x is *unsatisfiable*, where Ψ^q_x is polynomial in size of $\mathcal{E}(x)$ and exponential in $|\mathsf{RVar}(\mathcal{E}(x))|$ and $|\mathfrak{D}|$.

Example 3. To illustrate the SMT-encoding, consider the program *SecExp3* (shown in Figure 7), which is a fragment of $P^{254}_{\texttt{inlined}}$. Given a private input k, it returns two shares (x_7, x_9) such that $x_7 \oplus x_9 = k^3$. This program is made buggy for the illustration purpose: the procedure call *RefreshMasks*(z_0, z_1) at call-site 24 in P^{254} is removed. For each variable x_i, by applying the type system, we can deduce:

$$\boxed{\begin{array}{lll}
\vdash \mathcal{E}(x) : \tau_{\mathsf{uf}}; & \vdash \mathcal{E}(x_0) : \tau_{\mathsf{si}}; & \vdash \mathcal{E}(x_1) : \tau_{\mathsf{si}}; \\
\vdash \mathcal{E}(x_2) : \tau_{\mathsf{uk}}; & \vdash \mathcal{E}(x_3) : \tau_{\mathsf{uk}}; & \vdash \mathcal{E}(x_4) : \tau_{\mathsf{uf}}; \\
\vdash \mathcal{E}(x_5) : \tau_{\mathsf{uf}}; & \vdash \mathcal{E}(x_6) : \tau_{\mathsf{uk}}; & \vdash \mathcal{E}(x_7) : \tau_{\mathsf{uf}}; \\
\vdash \mathcal{E}(x_8) : \tau_{\mathsf{si}}; & \vdash \mathcal{E}(x_9) : \tau_{\mathsf{uf}}. &
\end{array}}$$

There are only three full computations (of x_2, x_3 and x_6) whose distribution types are unknown. Suppose $\mathfrak{D} = \{0, 1, 2, 3\}$, then $\Psi^q_{x_2}$ is

$$\begin{array}{l}
\left(\begin{array}{ll}
c_0 = ((k \oplus 0) \odot (k \oplus 0)) \odot 0 & \wedge \\
c'_0 = ((k' \oplus 0) \odot (k \oplus 0)) \odot 0 & \wedge \\
c_1 = ((k \oplus 1) \odot (k \oplus 1)) \odot 1 & \wedge \\
c'_1 = ((k' \oplus 1) \odot (k \oplus 1)) \odot 1 & \wedge \\
c_2 = ((k \oplus 2) \odot (k \oplus 2)) \odot 2 & \wedge \\
c'_2 = ((k' \oplus 2) \odot (k \oplus 2)) \odot 2 & \wedge \\
c_3 = ((k \oplus 3) \odot (k \oplus 3)) \odot 3 & \wedge \\
c'_3 = ((k' \oplus 3) \odot (k \oplus 3)) \odot 3 &
\end{array}\right) \bigwedge \\
\quad \left(\begin{array}{ll}
I_0 = (c = c_0) \ ?\ 1 : 0 & \wedge \\
I_1 = (c = c_1) \ ?\ 1 : 0 & \wedge \\
I_2 = (c = c_2) \ ?\ 1 : 0 & \wedge \\
I_3 = (c = c_3) \ ?\ 1 : 0 &
\end{array}\right) \bigwedge \\
\quad \left(\begin{array}{ll}
I'_0 = (c = c'_0) \ ?\ 1 : 0 & \wedge \\
I'_1 = (c = c'_1) \ ?\ 1 : 0 & \wedge \\
I'_2 = (c = c'_2) \ ?\ 1 : 0 & \wedge \\
I'_3 = (c = c'_3) \ ?\ 1 : 0 &
\end{array}\right) \bigwedge \\
((I_0 + I_1 + I_2 + I_3) - (I'_0 + I'_1 + I'_2 + I'_3) > (1 - q) \times 2^2)
\end{array}$$

The formula $\Psi^1_{x_2}$ (i.e., $q = 1$) is satisfiable, so we conclude that x_2 is leaky. We can also conclude that x_6 is perfectly masked, while x_3 is leaky. This cannot be done by type systems in literature.

4.3 Domain Specific Heuristics

We provide in this subsection three heuristics to facilitate both type inference and model-counting based reasoning.

4.3.1 Ineffective Variable Elimination

In cryptographic programs, masking and de-masking are mixed during computations. The values of some random variables in a computation may become ineffective (see below for formal definition) after de-masking, then these

```
SecExp3(k){
   r0 = $;
   r1 = $;
   x = k ⊕ r0;
   x0 = x ⊙ x;
   x1 = r0 ⊙ r0;
   x2 = x0 ⊙ r0;
   x3 = x1 ⊙ x;
   x4 = r1 ⊕ x2;
   x5 = x4 ⊕ x3;
   x6 = x0 ⊙ x;
   x7 = x6 ⊕ r1;
   x8 = x1 ⊙ r0;
   x9 = x8 ⊕ x5;
   return (x7, x9);
}
```

Fig. 7. SecExp3: A fragment of SecExp254.

Algorithm 2: Simplifying expression

```
1  Function SIMPLIFY($e$)
2      if $e$ is $\neg e'$ then
3          return $\neg$SIMPLIFY($e'$);
4      if $e$ is $e_1 \circ e_2$ then
5          $e$ =SIMPLIFY($e_1$)$\circ$SIMPLIFY($e_2$);
6      forall $r \in \mathsf{RVar}(e)$ do
7          if $\mathsf{SAT}(e \neq e[0/r])$==$\mathsf{No}$ then
8              if $\sum_{\circ\in\{\wedge,\odot,\times,\ll,\gg,\oplus,+\}} |\mathsf{Sub}^{\circ}_{r}(e)| \geq |\mathsf{Sub}^{\vee}_{r}(e)|$
               then
9                  $e$ =MUTATE($e, r, 0$);
10             else
11                 $e$ =MUTATE($e, r, \vec{1}$);
12     return $e$;

13 Function MUTATE($e, r, c$)
14     forall $e_1 \in \bigcup_{\circ\in\{\oplus,+\}} \mathsf{Sub}^{\circ}_{r}(e)$ with $e_1$ as $r \circ e_2$ or
       $e_2 \circ r$ do
15         $e = (c = \vec{1})\ ?\ e[(\vec{1} \circ e_2)/e_1] : e[e_2/e_1]$ ;
16     forall $e_1 \in \bigcup_{\circ\in\{\wedge,\times,\odot,\ll,\gg\}} \mathsf{Sub}^{\circ}_{r}(e)$ with $e_1$ as $r \circ e_2$
       or $e_2 \circ r$ do
17         if $c == 0$ then
18             $e$ =MUTATE($e[r/e_1], r, 0$);
19         else
20             $e = e[(\vec{1} \circ e_2)/e_1]$;
21     forall $e_1 \in \mathsf{Sub}^{\vee}_{r}(e)$ with $e_1$ as $r \vee e_2$ or $e_2 \vee r$ do
22         if $c == \vec{1}$ then
23             $e$ =MUTATE($e[r/e_1], r, \vec{1}$);
24         else
25             $e = e[e_2/e_1]$;
26     if $\neg x \in \mathsf{Sub}(e)$ then
27         $e$ =MUTATE($e[r_{\texttt{new}}/(\neg r)], x_{\texttt{new}}, \neg c$);
28     $e = e[c/r]$;
29     return $e$;
```

variables can be instantiated by any concrete values without changing the distribution of the computation, but can facilitate both type inference and model-counting based reasoning. Based on this observation, we present an algorithm to identify and eliminate such kind of variables.

Given a partial computation e that does not contain any procedure calls, a random variable $r \in \mathsf{RVar}(e)$ is *ineffective* in e if e and $e[c/r]$ are equivalent for any $c \in \mathfrak{D}$ while $e[c/r]$ contains less variables, where $e[c/r]$ is obtained from e by instantiated r with c. Otherwise, we say r is *effective* in e. We denote by $\mathsf{InEffR}(e)$ and $\mathsf{EffR}(e)$ the sets of ineffective and effective random variables in e, respectively.

A naive approach for computing all the effective variables of e is iteratively invoking a SAT solver for checking whether $e \neq e[c/x]$ is satisfiable or not for some constant $c \in \mathfrak{D}$, for each random variable $x \in \mathsf{RVar}(e)$, as $e \neq e[c/x]$ is satisfiable iff the variable x is effective in the computation e. However, this approach needs to invoke SAT solvers at least $|\mathsf{RVar}(e)|$ times. (Recall that the satisfiability problem of propositional formulae is NP-complete.) In order to improve the efficiency in practical applications, we use an alternative, simple approach: once it is known that x is an ineffective variable (i.e., $x \in \mathsf{InEffR}(e)$) in the computation e, we immediately replace the variable x by some concrete value $c \in \mathfrak{D}$ and simplify the resulting computation $e[c/x]$ by algebraic laws. Here, rather than choosing a concrete value $c \in \mathfrak{D}$ for the variable x randomly, we choose a specific value c based on the syntactic structure of the computation e, so that the computation e can be simplified as much as possible. The basic idea is that if a variable x is an ineffective variable in a computation e and the computation $x \circ e'$ is a sub-expression therein for some $\circ \in \{\wedge, \odot, \times, \ll, \gg\}$, then it gains to replace the variable x by the value 0, as $0 \circ e'$ resulting in the constant 0 which can be used to simplify the computation e further. Similarly, replacing the variable x by the value $\vec{1}$ (note that $\vec{1} := 1^n$) can simplify $x \vee e'$ into the constant $\vec{1}$, and replacing x by 0 simplifying $x \circ e'$ for $\circ \in \{\oplus, +\}$ into e'. In practice, several sub-expressions may co-exist. Our strategy is then to instantiate the variable x based on the number of such sub-expressions: If the computation e contains more sub-expressions of the form $x \circ e'$ and $e' \circ x$ ($\circ \in \{\wedge, \odot, \times, \ll, \gg, \oplus, +\}$) than those of the form $x \vee e'$ and $e' \vee x$, we instantiate the variable x by the value 0, otherwise by the value $\vec{1}$.

Algorithm 2 presents the pseudo-code, where $\mathsf{Sub}(e)$ denotes the set of all the sub-expressions in e, and $\mathsf{Sub}^{\circ}_{x}(e)$ for every operator $\circ$ and variable x denotes the set of sub-expressions $\{e' \in \mathsf{Sub}(e) \mid e' \text{ is in the form of } x \circ e_1 \text{ or } x \circ e_1 \text{ for some expression } e_1\}$. Given a computation e that does not contain any procedure calls, SIMPLIFY(e) computes an equivalent but simpler computation e' with $\mathsf{RVar}(e') = \mathsf{EffR}(e)$. In detail, if e is in the form of $\neg e'$, SIMPLIFY returns $\neg$SIMPLIFY(e') (Line 3). Otherwise if e is in the form of $e_1 \circ e_2$, then we replace the sub-expression e_i by SIMPLIFY(e_i) for $i \in \{1, 2\}$ (Line 5). In our implementation, in order to reduce the number of calls to SAT/SMT solvers, we adopt a lazy strategy, i.e., we replace the computation e_i by SIMPLIFY(e_i) only if it has been computed. As a result, each random variable $x \in \mathsf{RVar}(e)$ is checked at most once by verifying whether the logical formula $(e \neq e[0/x])$ is satisfiable or not (Line 7). If it is not satisfiable (i.e., $\mathsf{SAT}(e \neq e[0/x])$=$\mathsf{No}$ in Algorithm 2), $x \in \mathsf{InEffR}(e)$ and we then invoke the function MUTATE (Lines 9 and 11).

MUTATE(e, x, c) mutates e according to the concrete value c of the variable x using algebraic laws, where $e[e_2/e_1]$ denotes the computation obtained from e by replacing all the occurrences of e_1 with e_2. It is easy to verify that:

Theorem 5. For every computation e that does not contain any procedure calls, SIMPLIFY(e) is equivalent to e and

$\mathsf{RVar}(\textsc{Simplify}(e)) = \mathsf{EffR}(e)$. Moreover, $\textsc{Simplify}(e)$ invokes SAT solvers at most $\mathsf{RVar}(e)$ times.

Example 4. For instance, r is *not* a dominant variable in $(r \oplus y) \oplus r$, but is ineffective. Therefore, $(r \oplus y) \oplus r$ can be simplified into $(0 \oplus y) \oplus 0$ by instantiating r with 0. $(0 \oplus y) \oplus 0$ is further simplified into y by algebraic laws.

Remark that in Algorithm 2 we do not count the number of computations of the form $\neg x \circ e$ when choosing concrete values, and we only check random variables. This is because: (1) negation rarely occurs in masked programs according to benchmarks we found; (2) some of the random variables may become ineffective after de-masking while this is rarely the case for other (non-random) variables. Reducing the number of random variables gains most because our SMT-based method constructs logic formulae whose sizes are exponential in the number of bits of random variables (cf. Section 4.2).

Besides computing $\mathsf{EffR}(e)$, the function $\textsc{Simplify}(e)$ can also simplify e. It is complementary to the two heuristic rules: rule (Conv) of Barthe et al. [35] and elementary circuit transformations of Coron [39].

4.3.2 Dominated Subexpression Elimination

Recall that if r is a dominant (random) variable of a computation e, then the distribution of the values of $\mathcal{E}(x)_{\texttt{inlined}}(\sigma)$ is uniform for all possible assignments σ of formal arguments and variables in $\mathcal{E}(x)_{\texttt{inlined}}$ (cf. Proposition 1 and Proposition 2(1)). This means that e can be safely regarded as a random variable r. Based on this observation, for any partial computation e' that contains e, we can regard e as a random variable r when evaluating e' if r does not appear in $e'[r/e]$. In other words, if e is an r-dominated partial computation in e' and the variable r does not occur in $e'[r/e]$, we can safely reason on $e'[r/e]$ instead of e'.

For a given partial computation e, we denote by $\widehat{e}$ the computation obtained by iteratively applying ineffective variable and dominated subexpression eliminations on the computation e. Note that ineffective variable elimination can be applied only if e does not contain any procedure calls.

Lemma 3. For any variable x in the program P, $\mathcal{E}(x)_{\texttt{inlined}}(\sigma)$ and $\widehat{\mathcal{E}(x)}_{\texttt{inlined}}(\sigma)$ have the same distribution for any assignment σ of variables and formal arguments in $\mathcal{E}(x)_{\texttt{inlined}}$.

Example 5. Let us consider the variable x_6 in the program shown in Figure 7, where $\vdash \mathcal{E}(x_6) : \tau_{\mathsf{uk}}$ and $\mathcal{E}(x_6) = ((k \oplus r_0) \odot (k \oplus r_0)) \odot (k \oplus r_0)$. It is easy to see that $(k \oplus r_0)$ is r_0-dominated computation of $\mathcal{E}(x_6)$. Therefore, $\mathcal{E}(x_6)$ can be simplified into $\widehat{\mathcal{E}(x_6)} = r_0 \odot r_0 \odot r_0$. Consequently, we can deduce that $\vdash \mathcal{E}(x_6) : \tau_{\mathsf{si}}$ by applying rule (NoKey) on $\widehat{\mathcal{E}(x_6)}$. This avoids to invoke the expensive model-counting methods.

We also leverage dominated subexpression elimination to handle variables that are return values of function calls. Recall that for each variable x_i which is defined by $x_i = f(y_1, \cdots, y_m)[i]@\ell$; for some $1 \le i \le k$ after procedure inlining, we regard the partial computation $\mathcal{E}(x_i)$ (without inlining) as a special computation such that $\mathsf{RVar}(\mathcal{E}(x_i)) = \{r@\ell \mid r \in \mathsf{RVar}(\mathcal{E}(z_i))\} \cup RA$. For each computation e that uses x_i, when reasoning on the computation e, $\mathsf{RVar}(\mathcal{E}(x_i))$ will be refined to the set $\mathsf{Dom}(\mathcal{E}(x_i))$ if $\mathsf{Dom}(\mathcal{E}(x_i)) \cap \mathsf{RVar}(e[x_i'/x_i]) = \emptyset$, where x_i' denotes a fresh variable. Similarly, to check whether $\mathsf{Dom}(e) \setminus \mathsf{RVar}(e') \neq \emptyset$ and/or $\mathsf{RVar}(e) \cap \mathsf{RVar}(e') = \emptyset$, $\mathsf{RVar}(\mathcal{E}(x_i))$ will be refined to the set $\mathsf{Dom}(\mathcal{E}(x_i))$ if $\mathsf{Dom}(\mathcal{E}(x_i)) \cap \mathsf{RVar}(e[x_i'/x_i]) = \emptyset$ and $\mathsf{Dom}(\mathcal{E}(x_i)) \cap \mathsf{RVar}(e'[x_i'/x_i]) = \emptyset$.

Example 6. Let us consider the procedure *SecExp254* in our running example. Since z_{10} in the partial computation $\mathcal{E}(z_{12}) = z_{10}^{16}$ is a return value of the procedure call at the call-site 29 and $\mathsf{Dom}(\mathcal{E}(z_{10})) = \{r_0@29\}$, $\mathsf{RVar}(\mathcal{E}(z_{10})) = \{r_0@29, r_0@25, r_1@24, r_1@28\}$ can be refined to the set $\mathsf{RVar}(\mathcal{E}(z_{10})) = \{r_0@29\}$. Since $r_1@28 \in \mathsf{Dom}(\mathcal{E}(z_8))$, we have that $\mathsf{Dom}(\mathcal{E}(z_8)) \setminus \mathsf{RVar}(\mathcal{E}(z_{12})) \neq \emptyset$, hence *SecMult*$(z_{12}, z_{13}, z_8, z_9)$ satisfies the formula $\mathsf{Dom}(b_0) \setminus \mathsf{RVar}(a_0) \neq \emptyset$, which is a conjunct of the annotation ψ_1 of the procedure *SecMult*.

4.3.3 Transformation Oracle

In order to utilize human knowledge of cryptographic programs which can facilitate both type inference and model-counting based reasoning, we provide a mechanism called transformation oracle. The transformation oracle Ω is a set of 3-tuples of the form $(e, r, 1)$ and $(e_1, e_2, 0)$ such that

- if $(e, r, 1) \in \Omega$, then for every partial computation e' containing e and random variable r does not occur in $e'[r/e]$, e in e' can be replaced by r;
- if $(e_1, e_2, 0) \in \Omega$, then for every partial computation e' containing e_1, all the occurrences of e_1 in e' can be safely replaced by the partial computation e_2.

For instance, the tuple $(r \oplus ((2 \times r) \wedge e),\ r, 1)$ is used in our experiments.

For a given transformation oracle Ω and a partial computation e', we denote by $\Omega(e')$ the resulting computation after applying the transformation oracle Ω on e' when it is applicable.

4.4 Overall Algorithms

In this subsection, we present the overall algorithm for verifying perfect masking and computing QMS values, by leveraging the three key techniques presented in the preceding three subsections.

We denote by $[\![e]\!]_\Phi$ a pair (τ, cxt) consisting of the distribution type τ of the computation e obtained by type inference without using rule (Inline) and cxt is a flag indicating whether the assumption Φ is used during type inference (cxt being set $\texttt{True}$ means Φ is used). Rule (Inline) is used in an on-demand fashion. We also denote by $\mathsf{MCSolver}(e, q)$ the procedure of model-counting (cf. Section 4.2) which returns SAT if $\texttt{QMS}_x \ge q$ for $e = \mathcal{E}(x)$.

4.4.1 Algorithm for Perfect Masking Verification

Fix a non-recursive program P which uses the sets of public (X_p) and private (X_k) input variables. The overall procedure for perfect masking verification is shown in Algorithm 3.

We use the following data structures: Y_{lk} is a set storing all the internal variables of $P_{\texttt{inlined}}$ that are leaky, λ_t is a map that labels each variable with its distribution type, λ_{exp} is a map that records the computation of each internal

Algorithm 3: Perfect masking verification

```
1  Function PMCHECKING($P, X_p, X_k, \Omega$)
2      $Y_{lk} = \emptyset$; $\lambda_t = \lambda_{exp}$ =emptymap;
3      $Y^f_{cxt} = Y^f_{ukd} = \emptyset$ for each procedure $f$ of $P$;
4      foreach procedure $f$ of $P$ in a reverse topological order of the call graph $P$ do
5          CHECKPROC($f$);
6      return $Y_{lk}$;
7  Function CHECKPROC($f$)
8      $Todo = X^f \setminus X^f_r$;
9      foreach statement $s$ of $f$ from the first to the last do
10         if $s$ is $x = e$ and $x \in Todo$ then
11             $\lambda_{exp}(x) = \mathcal{E}(x)$;
12             $(\tau, cxt)$ =CHECKEXPR($\lambda_{exp}, x, \Phi_f$);
13             if $\tau \neq \tau_{\mathsf{uk}}$ then
14                 if $cxt ==$ True then
15                     $Y^f_{cxt} = Y^f_{cxt} \cup \{x\}$;
16                 $\lambda_t(x) = \tau$;
17             else if $\lambda_{exp}(x)$ contains a procedure call at the call-site $\ell$ then
18                 while $\lambda_{exp}(x)$ contains a procedure call at the call-site $\ell$ do
19                     $\lambda_{exp}(x) =$ inline($\lambda_{exp}(x), \ell$);
20                     $(\tau, cxt)$ =CHECKEXPR($\lambda_{exp}, x, \Phi_f$);
21                     if $\tau \neq \tau_{\mathsf{uk}}$ then
22                         if $cxt ==$ True then
23                             $Y^f_{cxt} = Y^f_{cxt} \cup \{x\}$;
24                         $\lambda_t(x) = \tau$;
25                         break;
26                 if $\tau == \tau_{\mathsf{uk}}$ then
27                     if $f \neq main$ then
28                         $Y^f_{ukd} = Y^f_{ukd} \cup \{x\}$;
29                     else
30                         if MCSolver($\lambda_{exp}(x), 1$) $\neq$SAT then
31                             $Y_{lk} = Y_{lk} \cup \{x\}$;
32                             $\lambda_t(x) = \tau_{\mathsf{lk}}$;
33                         else $\lambda_t(x) = \tau_{\mathsf{si}}$;
34         else if $s$ is $x_1, \cdots, x_k = g(y_1, \cdots, y_m)$ at $\ell$ then
35             if $g(y_1, \cdots, y_m) \models \Phi_g$ then
36                 if $Y^g_{ukd} \neq \emptyset$ then
37                     $Todo = Todo \cup \{y@\ell \mid y \in Y^g_{ukd}\}$;
38                     $f =$ inline($f, \ell$);
39             else
40                 $Todo = Todo \cup \{y@\ell \mid y \in Y^g_{cxt} \cup Y^g_{ukd}\}$;
41                 $f =$ inline($f, \ell$);
42     return;
43 Function CHECKEXPR($\lambda, x, \Phi$)
44     if $[\![\lambda(x)]\!]_\Phi[0] \in \{\tau_{\mathsf{si}}, \tau_{\mathsf{uf}}, \tau_{\mathsf{lk}}\}$ then
45         return $[\![\lambda(x)]\!]_\Phi$;
46     $\lambda(x) = \widehat{\lambda(x)}$;
47     if $[\![\lambda(x)]\!]_\Phi[0] \in \{\tau_{\mathsf{si}}, \tau_{\mathsf{uf}}, \tau_{\mathsf{lk}}\}$ then
48         return $[\![\lambda(x)]\!]_\Phi$;
49     if $\exists\Omega(\lambda(x)) : [\![\Omega(\lambda(x))]\!]_\Phi[0] \in \{\tau_{\mathsf{si}}, \tau_{\mathsf{uf}}, \tau_{\mathsf{lk}}\}$ then
50         return $[\![\Omega(\lambda(x))]\!]_\Phi$;
51     return $(\tau_{\mathsf{uk}}$, True);
```

variable, $Y^f_{cxt} \subseteq X^f$ stores the variables whose distribution types are τ_{si} or τ_{uf} under the assumption Φ_f, and $Y^f_{ukd} \subseteq X^f$ stores the variables whose distribution types are τ_{uk}.

The function PMCHECKING (in Algorithm 3) checks whether P is perfectly masked. After initialization (Lines 2 and 3), it checks each procedure in a reverse topological order of the call graph of P by invoking the function CHECKPROC (Line 5). Recalling that P is non-recursive, reverse topological order ensures that all the called procedures in f have been verified when checking the procedure f.

The function CHECKPROC infers the distribution types of internal variables of the given procedure f. It first initializes the set *Todo* storing the internal variables whose computations should be verified (Line 8). Then, it iteratively traverses statements in the procedure f (Line 9). For each statement s, it works as follows. (Note that if s is in the form of $r = \$$, then r is a random variable and must have type τ_{uf}, hence such statements are skipped.)

1) If s is in the form of $x = e$ and $x \in Todo$ (Line 10), then the partial computation $\mathcal{E}(x)$ is constructed and stored in λ_{exp} (Line 11). It first applies the type system to infer its distribution type by invoking the function CHECKEXPR (Line 12), which returns a pair (τ, cxt), where τ denotes the distribution type of $\lambda_{exp}(x)$ and cxt is a flag indicating whether the assumption Φ_f is used or not during type inference.
   a) If the type of $\lambda_{exp}(x)$ is not τ_{uk} (Line 13), then the type of x is recorded in λ_t (Line 16). Moreover, if the assumption Φ_f is used during type inference, then x is added into the set Y^f_{cxt} (Line 15) which will be verified again when a procedure call to f does not satisfy the assumption Φ_f. In other words, variables that can be proved having type τ_{si} or τ_{uf} or τ_{lk} without using the assumption Φ_f will not be verified again even if Φ_f is not satisfied by procedure calls to f.
   b) If $\lambda_{exp}(x)$ has type τ_{uk} and contains a procedure call at the call-site ℓ (Line 17), then $\lambda_{exp}(x)$ is updated by inlining the procedure call at ℓ (Line 19) and continues inferring distribution type $\lambda_{exp}(x)$, i.e., step 1).
   c) If $\lambda_{exp}(x)$ has type τ_{uk} and does not contain any procedure call and f is not the *main* procedure (Lines 26–27), x is added into the set Y^f_{ukd} (Line 28) which will be verified again when the procedure f is inlined no matter the assumption Φ_f is satisfied or not.
   d) If $\lambda_{exp}(x)$ has type τ_{uk} and does not contain any procedure call and f is the *main* procedure (Line 29), then we apply the model-counting based methods by invoking MCSolver($\lambda_{exp}(x), 1$) (Line 30). There are two possible outcomes: $\lambda_{exp}(x)$ is τ_{si} or τ_{lk}. The type is also stored in λ_t (Lines 32 and 33). Moreover, if the type is τ_{lk}, then x is added into Y_{lk}, i.e., x is leaky. The update of the type of $\lambda_{exp}(x)$ might facilitate the type inference for fan-out computations of x.
2) If s is a procedure call $x_1, \cdots, x_k = g(y_1, \cdots, y_m)$ at call-site ℓ (Line 34), then it checks whether the procedure call $g(y_1, \cdots, y_m)$ satisfies the assumption Φ_g.
   a) If $g(y_1, \cdots, y_m)$ satisfies Φ_g and Y^g_{ukd} is nonempty (Lines 35–36), then the procedure call at ℓ is inlined and the variables $y@\ell$ for $y \in Y^g_{ukd}$ are added into *Todo* for rechecking (Line 37). We emphasize that when $g(y_1, \cdots, y_m)$ satisfies Φ_g and Y^g_{ukd} is empty, the whole procedure call $x_1, \cdots, x_k = g(y_1, \cdots, y_m)$ can be skipped without inlining, as the distribution types of variables $y@\ell$ and y are the same for each $y \in X^g$.
   b) If $g(y_1, \cdots, y_m)$ does not satisfy Φ_g (Line 39), the procedure call $x_1, \cdots, x_k = g(y_1, \cdots, y_m)$ at ℓ is inlined and the variables $y@\ell$ for $y \in Y^g_{ukd} \cup Y^g_{cxt}$ are

added into *Todo* for rechecking (Line 40). Note that variables $y \in X^g \setminus (Y^g_{ukd} \cup Y^g_{cxt})$ are not added into *Todo*, as the distribution types of their computations can be inferred without using the assumption Φ_g.

To check whether a procedure call $g(y_1, \cdots, y_m)$ satisfies the assumption Φ_g in Algorithm 3, we iteratively check whether the procedure call $g(y_1, \cdots, y_m)$ satisfies some formula $\psi \in \Phi_g$, which is done by iteratively checking each conjunct of ψ according to the satisfaction relation defined in Section 4.1.3. To check $g(y_1, \cdots, y_m) \models \mathsf{Dom}(a_i) \setminus \mathsf{RVar}(a_j) \neq \emptyset$ and/or $g(y_1, \cdots, y_m) \models \mathsf{RVar}(a_i) \cap \mathsf{RVar}(a_j) = \emptyset$. we leverage Proposition 2 which may allow to get the conclusive result without constructing (cf. Section 4.1.6).

Theorem 6. Given a non-recursive program P, by Algorithm 3, $Y_{lk} = \emptyset$ iff P is perfectly masked. Moreover, if $x@\ell_k \cdots @\ell_1 \in Y_{lk}$, then the internal variable $x@\ell_k \cdots @\ell_1$ is leaky, where $\ell_1, \cdots, \ell_k$ is the sequence of call-sites to reach the procedure that contains x.

By disabling model-counting in Algorithm 3 and interpreting all τ_{uk}-typed variables as potentially flaws, Algorithm 3 degenerates to a sound type inference procedure, which is fast and potentially more accurate than those in [35], [43], [60], [61], owing to the heuristics introduced in Section 4.3 and the type system supporting compositional reasoning.

4.4.2 Algorithm for QMS Computing

To quantify resistance of a program, we present a binary search based function QMSComputing (in Algorithm 4) to compute QMS values. QMSComputing first invokes the function PMChecking (in Algorithm 3) to perform perfect masking verification (Line 2). Then, it checks for each variable $x \in \texttt{Leakpoints}$. For each variable $x \in \texttt{Leakpoints}$ whose computation $\lambda_{exp}(x)$ does not contain any random variable, we directly deduce that $\texttt{QMS}_x = 0$ (Line 5). Otherwise if $\lambda_{exp}(x)$ contains some random variables, we use either the brute-force method or an SMT-based binary search to compute $\texttt{QMS}_x$ based on the following observation:

$$\texttt{QMS}_x = \frac{i}{2^{n \times |\mathsf{RVar}(\lambda_{exp}(x))|}}, \text{ for some } 0 \leq i \leq 2^{n \times |\mathsf{RVar}(\lambda_{exp}(x))|}.$$

The while-loop in Algorithm 4 (Lines 9–15) executes at most $\mathbf{O}(n \times |\mathsf{RVar}(\lambda_{exp}(x))|)$ times for each x, hence Algorithm 4 always terminates.

5 Implementation and Evaluation

We have implemented our approach in a verification tool QMVerif, which uses Z3 [47] as the underlying SMT solver with fixed size bit-vector theory. We conduct experiments on both Boolean and arithmetic programs including various implementations of full AES, DES and MAC-Keccak.

The experiments are designed to answer the following research questions (RQs):

RQ1. How effective and efficient is the type inference algorithm on arithmetic programs with procedure calls?

RQ2. How is the overall approach performed on arithmetic programs (without procedure calls), compared with EasyCrypt [35]?

Algorithm 4: Computing QMS

```
1  Function QMSComputing($P, X_p, X_k, \Omega$)
2      Leakpoints =PMChecking($P, X_p, X_k, \Omega$);
3      foreach $x \in$ Leakpoints do
4          if $\mathsf{RVar}(\lambda_{exp}(x)) == \emptyset$ then
5              $\texttt{QMS}_x = 0$;
6          else
7              low = 0;
8              high = max = $2^{n \times |\mathsf{RVar}(\lambda_{exp}(x))|}$;
9              while low < high do
10                 mid = $\lceil \frac{\texttt{low}+\texttt{high}}{2} \rceil$;
11                 $q = \frac{\texttt{mid}}{\texttt{max}}$;
12                 if MCSolver($\lambda_{exp}(x), q$) $\neq$SAT then
13                     high := mid − 1
14                 else
15                     low = mid;
16             $\texttt{QMS}_x = \frac{\texttt{low}}{\texttt{max}}$;
17     return;
```

TABLE 1
Variant versions of Sbox implementations, where column *Order* denotes the masking order, column *Refresh* denotes the procedure *Refresh*, column *SecMult* denotes the procedure *SecMult*, column *Power254* denotes the procedure *Power254*.

	Order		Refresh		SecMult		Power254	
	1st	2nd	[28]	[14]	[28]	[62]	[28]	[33]
Sbox1	✓		✓			✓	✓	
Sbox2	✓		✓		✓		✓	
Sbox3	✓				✓			✓
Sbox4	✓					✓		✓
Sbox5		✓	✓			✓	✓	
Sbox6		✓	✓		✓		✓	
Sbox7		✓		✓		✓	✓	
Sbox8		✓		✓	✓		✓	
Sbox9		✓			✓			✓
Sbox10		✓				✓		✓

RQ3. How is the overall approach performed on Boolean programs (without procedure calls), compared with state-of-the-art tools QMSInfer [44], SC Sniffer [42], [50] and maskVerif [37]?

In all experiments, we used a machine with Intel Xeon E5-2690v4 2.6GHz CPU, 64-bit Ubuntu 16.04.4 LTS operating system, and 256 GB RAM (only one core is used in our computation).

5.1 RQ1: Experiments on Arithmetic Programs with Procedure Calls

To address **RQ1**, we implemented 10 versions of Sbox based on the algorithm in [33] by varying the underlying sub-procedures as shown in Table 1. Column 1 gives the benchmark name. Columns 2-3 show the masking order (note that we only verify first-order even for second-order benchmarks). Columns 4-5 show the two variants of Refresh functions: where the first one is addition-based mask refreshing algorithm from [28], and the second one is a multiplication-based mask refreshing algorithm from [14]. Columns 6-7 show the two variants of SecMult functions, where the first one is a SecMult algorithm with $\mathcal{O}(n^2)$ memory from [28] and the second one is the improvement of the first which is a linear memory algorithm proposed in [62].

TABLE 2
Results of perfect masking verification under the HW leakage model, where *Pre-inlined* means that all the procedure calls are inlined before verification, *No-assumption* means that all the procedures are only annotated by $\top$, *One-assumption* means that all the procedures of AES except for *SecMult* and all the procedures of DES except for *Sbox* are annotated by $\top$, *All-assumptions* means that all the procedures are annotated by well-designed assumptions, column *Name* gives the benchmark name, columns labeled by ♯*Checked* give the number of internal variables that are checked, columns labeled by *Time(s)* show the running time of verification in second, columns ♯*Inlining* show the number of times procedure inlining was performed.

Name	Pre-inlined			No-assumption			One-assumption			All-assumptions		
	♯Checked	Time(s)	♯Inlining	♯Checked	Time(s)	♯Inlining	♯Checked	Time(s)	♯Inlining	♯Checked	Time(s)	♯Inlining
Sbox1	46	≈0	8	47	≈0	8	35	≈0	4	19	≈0	0
Sbox2	50	≈0	8	47	≈0	8	35	≈0	4	19	≈0	0
Sbox3	70	≈0	6	141	≈0	6	141	≈0	6	60	≈0	4
Sbox4	68	≈0	6	141	≈0	6	141	≈0	6	60	≈0	4
Sbox5	110	≈0	8	104	≈0	8	77	≈0	5	42	≈0	2
Sbox6	122	≈0	8	104	≈0	8	77	≈0	5	42	≈0	2
Sbox7	118	≈0	8	116	≈0	8	89	≈0	5	50	≈0	2
Sbox8	130	≈0	8	116	≈0	8	89	≈0	5	50	≈0	2
Sbox9	178	≈0	6	317	≈0	6	317	≈0	6	150	≈0	3
Sbox10	172	≈0	6	317	≈0	6	317	≈0	6	150	≈0	3
AES1	11,142	314.5	2,632	3,678	1.4	362	3,666	0.2	358	2,182	0.1	320
AES2	11,942	196.9	2,632	3,678	1.4	362	3,666	0.3	358	2,183	0.1	320
AES3	15,942	558.6	2,232	6,570	274.1	387	6,570	274.8	387	2,393	0.1	338
AES4	15,542	559.6	2,232	6,570	292.5	387	6,570	293.6	387	2,392	0.1	338
AES5	24,724	2,669.8	2,632	6,501	7.7	362	6,474	0.5	359	4,214	0.4	345
AES6	27,124	3,504.7	2,632	6,501	7.7	362	6,474	0.6	359	4,214	0.4	345
AES7	26,324	2,932.8	2,632	6,991	8.7	362	6,964	0.6	359	4,430	0.5	345
AES8	28,724	3,129.8	2,632	6,991	9.3	362	6,964	0.6	359	4,430	0.5	345
AES9	38,324	2,928.6	2,232	12,823	1,268.1	387	12,823	1,286.3	387	3,786	0.2	338
AES10	37,124	3,064.1	2,232	12,823	1,252.7	387	12,823	1,266.6	387	3,786	0.2	338
DES1	82,304	327.3	6,450	74,507	289.8	458	50,571	257.0	458	38,288	222.4	516
DES2	39,552	81.4	4,914	23,237	26.3	446	16,165	16.6	446	7,748	14.1	549
DES3	248,448	53.6	3,122	157,618	287.3	432	22,450	20.0	432	6,602	7.4	491
DES4	215,680	86.7	3,122	85,681	153.0	432	20,145	18.2	432	6,345	13.4	491

Columns 8-9 show the two Power254 functions, where the first one is from [28] which needs mask refreshing and the second is from [33] which does not need mask refreshing.

We also implemented 10 versions of AES based on the algorithm in [28] by varying the underlying Sbox implementations and 4 versions of DES based on the existing implementations[1], where AESi uses the Sbox$i$ from Table 1, DES1 is a countermeasure with the Parity-Split method of Sbox computation which requires 10 non-linear multiplications from [63], DES2 is an improved method from [64] which requires only 4 non-linear multiplications, DES3 is based on the table recomputation [62], and DES4 [65] is a variant of, but twice as efficient as, DES3.

We verify these benchmarks using Algorithm 3 (i.e., perfect masking verification under the HW model), but excluding the model-counting methods. In order to gain insights on the assume-guarantee based compositional reasoning, we conduct experiments under four different settings:

1) Pre-inlined: all the procedure calls are inlined in advance;
2) No-assumption: all the procedures are only annotated by $\top$, but are inlined on-demand;
3) One-assumption: all the procedures of AES except for *SecMult* and all the procedures of DES except for *Sbox* are annotated by $\top$, but are inlined on-demand;
4) All-assumptions: all the procedures are annotated by well-designed assumptions.

Results. The experimental results are reported in Table 2, where Columns 2-4 (resp. Columns 5-7, Columns 8-10 and Columns 11-13) show the number of internal variables that have been checked (note that variables of the form $x@\ell_k\cdots@\ell_1$ that appear after procedure inlining are regarded as new internal variables instead of the variable x), the running time of verification (in second), and the number of times procedure inlining was performed. Note that the "No-assumption" setting corresponds to the type system proposed in the preliminary version of this paper [1].

Overall, our type inference algorithm is highly effective and efficient on programs with annotated assumptions. All the programs of AES can be proved secure in less than 1 second and all the programs of DES can be proved secure in less than 4 minutes (3 out of 4 were done in less than 15 seconds). By comparing Columns 2-4 with Columns 11-13, we observe that the compositional reasoning significantly reduces the number of times procedure inlining was performed, hence reducing the number of internal variables that have to be checked, and verification time (on AES family of programs, there are 3-4 orders of magnitude reductions).

We can also observe from Columns 5-10 that at large our type inference algorithm is also effective on large programs (AES1–AES10 and DES1–DES4) that do not have any assumptions or have only one procedure annotated with assumption. (Some exceptions include DES3 and DES4 under the "No-assumption" setting, the reason of which will be explained below.) This demonstrates the significance of on-demand procedure inlining.

[1] https://github.com/coron/htable.

TABLE 3
Results of perfect masking verification and computing QMS values on masked arithmetic programs under the HW leakage model, where column *Description* gives the name and reference of the program, column *Result* gives the ground truth (✓ for leakage-free and ✗ for the opposite), column $|X_i|$ denotes the number of internal variables, column $\sharp\tau_{\mathsf{lk}}$ denotes the number of leaky internal variables, column ♯*ModelCounting* denotes the number of internal variables which need model-counting methods, column *SMT* denotes the verification time using the SMT-based method as the model-counting method, column *B.F.* denotes the verification time using the brute-force method as the model-counting method, column *value* shows the QMS values of all leaky variables (note that duplicated values are omitted), column *EasyCrypt* replicates the total running time reported by Barthe et al. [35], and (12) in column *SMT* means that Z3 emits segmentation fault after verifying 12 internal variables.

Description	Result	$\lvert X_i \rvert$	$\sharp\tau_{\mathsf{lk}}$	Perfect Masking Verification			QMS			EasyCrypt [35]
				♯ModelCounting	SMT	B.F.	SMT	B.F.	Value	Time
SecMult [28]	✓	11	0	0	≈0s	≈0s	-	-	1	≈0s
Sbox (4) [33]	✓	66	0	0	≈0s	≈0s	-	-	1	≈0s
AES (4) [33]	✓	20,060	0	0	**≈2s**	**≈2s**	-	-	1	128s
MAC-Keccak	✓	18,218	0	0	**≈83s**	**≈83s**	-	-	1	405s
B2A [16]	✓	8	**0**	**1**	17s	**2s**	-	-	1	
A2B [16]	✓	46	0	0	≈0s	≈0s	-	-	1	
B2A [17]	✓	82	0	0	≈0s	≈0s	-	-	1	
A2B [17]	✓	41	0	0	≈0s	≈0s	-	-	1	
B2A [18]	✓	11	**0**	**1**	**1m 35s**	10m 59s	-	-	1	
B2A [19]	✓	16	0	0	≈0s	≈0s	-	-	1	
Sbox [28]	✓	45	0	0	≈0s	≈0s	-	-	1	
Sbox [27]	✗	772	**2**	**1**	≈0s	≈0s	0.9s	≈0s	0	
k^3	✗	11	2	2	96m 59s	**0.2s**	>4d	32s	0.988	
k^{12}	✗	15	2	2	101m 34s	**0.3s**	>4d	27s	0.988	
k^{15}	✗	21	4	4	93m 27s (12)	**28m 17s**	>4d	≈64h	0.988, 0.98	
k^{240}	✗	23	4	4	93m 27s (12)	**30m 9s**	>4d	≈64h	0.988, 0.98	
k^{252}	✗	31	4	4	93m 27s (12)	**32m 58s**	>4d	≈64h	0.988, 0.98	
k^{254}	✗	39	4	4	93m 27s (12)	**30m 9s**	>4d	≈64h	0.988, 0.98	

One may notice that the effectiveness and efficiency vary in benchmarks under different settings, namely, (1) the assumption of *SecMult* does not reduce the number of procedure inlines on Sbox3, Sbox4, Sbox9, Sbox10, AES3, AES4, AES9 and AES10, compared with the "No-assumption" setting, but it does reduce the number of procedure inlines on the other Sbox and AES benchmarks; (2) the verification time on DES3 and DES4 under "No-assumption" setting is greater than the one under "Pre-lined" setting although the number of procedure inlines is reduced; and (3) the number of procedure inlines on DES1–DES4 under the "All-assumptions" setting is greater than the one under the "No-assumption" and "One-assumption" settings, but the verification time is reduced.

To explain this observation, an in-depth analysis reveals that: For observation (1), all the benchmarks Sbox3, Sbox4, Sbox9, Sbox10, AES3, AES4, AES9 and AES10 use the *Power254* procedure in Sbox (cf. Table 1), while there are some τ_{uk}-typed variables in *Power254*, which are not resolved after inlining the procedure calls to *Power254* in Sbox. These τ_{uk}-typed variables are proved secure eventually in the *main* procedure. Consequently, these τ_{uk}-typed variables have to be checked multiple times. This problem is avoided when more procedure assumptions are provided, as shown under the "All-assumptions" setting. Observation (2) follows similar explanation as in observation (1). For observation (3), each procedure call is inlined only once under the "Pre-inlined" setting, while some procedure calls may be inlined multiple times under the other settings. For instance, consider an internal variable x whose partial computation $\mathcal{E}(x)$ relies upon some return values of several procedure calls to f, while the partial computation of these return values also relies upon the return values of another procedure call g. In this case, the same procedure call to g will be inlined once for each procedure call to f, resulting in multiple times of procedure inlines in partial computations. This limitation could be avoided by directly inlining the procedure calls in procedures instead of partial computations. We do not use this strategy, as we found that the verification time mainly depends on the number of internal variables to be checked rather than the number of procedure inlines. Moreover, inlining some procedure calls may be unnecessary and could increase the size of partial computations.

5.2 RQ2: Experiments on Arithmetic Programs without Procedure Calls

To address RQ2, we use the first-order masked arithmetic programs provided by the authors of [35], which are secure multiplication (SecMult) [28], Sbox [28], [33], full AES [33], full MAC-Keccak. In addition, we implemented the conversion algorithms from Boolean to arithmetic maskings (B2A) [16], [17], [18], [19], conversion algorithms from arithmetic to Boolean maskings (A2B) [16], [17], and buggy fragments $k^3, \ldots, k^{254}$ of first-order secure exponentiation [28] without the first RefreshMask function. For all the programs, we set $\mathfrak{D} = \{0, \cdots, 2^8 - 1\}$.

We conduct experiments of perfect masking verification and of computing QMS values, under both the HW and HD leakage models.

5.2.1 *Perfect Masking Verification under HW model*

The experimental results of perfect masking verification under the HW leakage model are reported in Table 3. Column 1 gives the name and reference of the program. Column 2 shows the ground truth. Column 3 shows the number of internal variables. Column 4 shows the number

TABLE 4
Results of perfect masking verification and computing QMS values on masked arithmetic programs under the HD leakage model, where column *Description* gives the name and reference of the original program, column *Result* gives the ground truth (✓ for leakage-free and ✗ for opposite), column $|X_i|$ denotes the number of internal variables in the original program, column ♯*D* denotes the number of introduced dummy variables in the modified program, column $|X_i|$+♯*D* denotes the total number of internal variables in the modified program, column ♯τ_{lk} denotes the number of leaky internal variables of the modified program, column ♯*ModelCounting* denotes the number of internal variables which need model-counting, column ♯*Running Out-of-time* denotes the number of internal variables on which QMVERIF runs out of time (threshold=15 minutes per variable), column *Time* denotes the total running time for each modified program, and column *min(Value)* gives the minimum one of QMS values.

Description	Result	$\lvert X_i\rvert$	♯D	$\lvert X_i\rvert$+♯D	♯τ_{lk}	Perfect Masking Verification			QMS		
						♯ModelCounting	♯Out-of-time	Time	♯Out-of-time	Time	min(Value)
SecMult [28]	✓	11	3	14	0	0	0	0.1s	-	-	1
Sbox (4) [33]	✗	66	43	109	2	4	2	30m 25s	1	15m 23s	0.99
B2A [16]	✓	8	4	12	0	1	0	2s	-	-	1
A2B [16]	✓	46	41	87	0	25	19	327m 44s	-	-	1
B2A [17]	✓	82	46	128	0	0	0	0.1s	-	-	1
A2B [17]	✓	41	14	55	0	0	0	0.1s	-	-	1
B2A [18]	✓	11	1	12	0	1	0	11m	-	-	1
B2A [19]	✓	16	3	19	0	1	1	15m 1s	-	-	1
Sbox [28]	✓	45	31	76	0	0	0	0.1s	-	-	1
Sbox [27]	✗	772	511	1283	2	1	0	0.1s	0	0.1s	0
k^{12}	✗	15	4	19	2	2	0	0.3s	0	22.1s	0.988
k^{15}	✗	21	9	30	4	6	2	47m 21s	2	30m 25s	0.988
k^{240}	✗	23	11	34	4	6	2	47m 22s	2	30m 25s	0.988
k^{252}	✗	31	18	49	4	8	4	75m 40s	2	30m 26s	0.988
k^{254}	✗	39	25	64	4	8	4	77m 41s	2	30m 25s	0.988

of leaky variables. Column 5 shows the number of variables for which the model-counting based methods are needed. Columns 6-7 respectively show the total running time of our tool QMVERIF using SMT-based and brute-force methods. For comparison purpose, in Column 11, we replicate the total running time reported by Barthe et al. [35] on the common benchmarks, i.e., the first four programs in Table 3. The machine used there was a headless VM with a dual core 64-bit processor clocked at 2GHz (only one core is used in the computation). Their tool is a type based proof system which is sound but incomplete. (Remark that, to our knowledge, there is no open-source tool for automatically verifying masking countermeasure of arithmetic programs under the HW/HD leakage model.)

The experimental results show that: (1) almost all the internal variables can be proved leakage-free using our type system; (2) some internal variables cannot be proved leakage-free by our type system (e.g., in B2A [16], B2A [18] and Sbox [27], meaning that the type inference is inconclusive in these cases), but can be resolved by our model-counting based methods; (3) on the programs (except B2A [18]) where the model-counting based methods is needed (i.e., ♯Count is non-zero), the brute-force method is significantly faster than the SMT-based one. In particular, on programs $k^{15}, \ldots, k^{254}$, Z3 crashed with segmentation fault after verifying 12 internal variables in 93 minutes, while the brute-force method comfortably returns the results. After a manual examination of these programs, we found that the computations of τ_{uk}-typed variables (where the brute-force method is more efficient) involve finite-field multiplication ($\odot$), while the computation of the τ_{uk}-typed variable in B2A [28] (where the SMT-based method is more efficient) only use the exclusive-or ($\oplus$) operations and one subtraction ($-$) operation. This gives an empirical suggestion on which model-counting method should be selected.

One may notice that the verification time (0.2s) of AES (4) [33] is significantly shorter than the results in Table 2 under the “Pre-inlined” setting. After an in-depth analysis of the source code provided by the authors of [35], we found errors in the implementation of the *AddRoundKey* procedure so that many of internal variables can be quickly proved. We have informed authors of [35].

Compared with the tool of Barthe et al. [35] which also verified the first four programs, the performance of small programs SecMult [28] and Sbox (4) [33] is comparable, but on larger programs AES [33] and MAC-Keccak, our tool is significantly (4.8 and 64 times) faster than their tool.

It is important to mention that the transformation oracle is only used for verifying the program A2B [16]. In theory, model-counting based methods could be able to verify the program A2B [16], unfortunately, both the SMT-based and brute-force methods failed to terminate in 3 days. We also notice that the brute-force method had verified more internal variables than the SMT-based one. For instance, on the computation $((2 \times r_1) \oplus (x - r) \oplus r_1) \wedge r$ where x is a private input and r, r_1 are random variables, the brute-force method successfully verified in a few minutes, but the state-of-the-art SMT solver Z3 could not terminate in 2 days. We also tried another SMT solver Boolector [66] which is the winner of SMT-COMP 2018 on QF-BV, Main Track. It also failed to terminate in 3 days. Undoubtedly more systematic experiments are required in the future, but our results suggest that, contrary to the common belief, currently SMT-based approaches are not promising, which calls for more scalable techniques for domain-specific constraints.

5.2.2 Computing QMS Values under HW model

The experimental results of computing QMS values are reported in Table 3. Column 8 shows the time of the SMT-based method. Column 9 shows the time of the brute-force method. Column 10 shows the QMS values of all leaky variables (note that duplicated values are omitted). We only reported the time for computing QMS values here, while the time for perfect masking verification is *excluded*. We remark

TABLE 5
Results of perfect masking verification on Boolean programs, where column *Name* gives the name of the program, column *Result* gives the ground truth (✓ for leakage-free and ✗ for opposite), column $|X_i|$ denotes the number of internal variables, column $\sharp\tau_{\mathsf{lk}}$ denotes the number of leaky internal variables, column *♯ModelCounting* denotes the number of internal variables which need model-counting, column *SMT* denotes the results of applying the SMT-based method, column *B.F.* denotes the results of applying the brute-force method, the last four columns respectively show the total verification time of the tools QMSInfer [44], SC Sniffer [42] and maskVerif [37].

Name	Result	$\|X_i\|$	$\sharp\tau_{\mathsf{lk}}$	QMVerif			QMSInfer [44]	SC Sniffer [42]	maskVerif [37]	
				♯ModelCounting	SMT	B.F.	Time	Time	♯ModelCounting	Time
P12	✓	197k	0	0	2.9s	**2.7s**	3.8s	68m 3s	0	99m 4s
P13	✗	197k	4.8k	4.8k	2m 8s	**2m 6s**	38m 53s	70m 13s	1	2m 15s
P14	✗	197k	3.2k	3.2k	1m 58s	**1m 45s**	42m 44s	86m 58s	1	19m 52s
P15	✗	198k	1.6k	3.2k	**2m 25s**	2m 43s	44m 12s	93m 38s	N/A	N/A
P16	✗	197k	4.8k	4.8k	1m 50s	**1m 38s**	48m 20s	91m 02s	1	2m 18s
P17	✗	205k	17.6k	12.8k	1m 24s	**1m 10s**	81m 1s	248m 34s	1	2m 37s

there is no tool for computing QMS values of arithmetic programs, so no baseline is given there.

The experimental results show that: (1) the brute-force method is effective in computing QMS values, but it is less efficient comparing to perfect masking verification: it takes roughly 64 hours on the programs k^{15}, k^{240}, k^{252} and k^{254}; (2) the brute-force method is also more efficient than the SMT-based method for computing QMS values; (3) the SMT-based method is only able to compute the QMS value of the leaky variable in Sbox [28], but fails for the others after 4 days. Indeed, Z3 cannot even finish the first iteration of the binary search on the smallest formula in 4 days. This, again, indicates the ineffectiveness of current SMT-based approaches. We manually examine $k^3, ..., k^{254}$ programs and find out that (1) variables used in the computations $\mathcal{E}(x)$ of leaky variables x are the same, and (2) the computations that can be quickly verified contain at most 4 operations, while the others contain at least 19 operations.

5.2.3 Perfect Masking Verification under HD model

In order to conduct experiments under the HD leakage model, we collect a set of variable pairs for each program. For each variable pair, we add a dummy variable as discussed in Section 2.2. The experimental results of QMVerif with the brute-force method enabled are reported in Table 4. Column 1 shows the program under consideration in which dummy variables are added. Column 2 gives the ground truth. Columns 3–5 show the numbers of original internal variables, dummy variables, and the total number of internal variables. Column 6 is the number of τ_{lk} variables. Column 7 is the number of variables for which the brute-force method is invoked. Column 8 is the number of variables on which the verification runs out of time (15 minutes per variable). Column 9 is the total running time of verification.

We can observe that: (1) many programs that are secure under the HW leakage model are still secure under the HD leakage model; and (2) almost all the dummy variables can be solved using type inference, while some dummy variables do need to invoke the brute-force model-counting method; (3) some variables cannot be verified in 15 minutes. We remark that no transformation oracle can be applied on many dummy variables in A2B [16], which may explain that the verification of these variables runs out of time.

5.2.4 Computing QMS Values under HD model

We conduct experiments of computing QMS values under the HD model on the modified arithmetic programs from Section 5.2.3. The experimental results of QMVerif with the brute-force method enabled are shown in the last three columns of Table 4, where Column 10 shows the number of internal variables on which QMVerif runs out of time for computing QMS values (15 minutes per variable); Column 11 shows the running time for computing QMS values excluding the time of perfect masking verification; and Column 12 shows the minimum one of QMS values.

The experimental results show that (1) QMVerif is able to compute the QMS values of most leaky variables under the HD leakage model; and (2) QMVerif fails to compute QMS values in 15 minutes for some leaky variables, due to the large size of computations (more than 20 operations per computation).

5.3 RQ3: Experiments on Boolean Programs without Procedure Calls

To address RQ3, we collect Boolean programs from the publicly available cryptographic software implementations [41]. There are 17 Boolean programs (P1–P17). We choose the programs P12–P17, which are the regenerations of MAC-Keccak reference code submitted to the SHA-3 competition held by the US National Institute of Standards and Technology. (The other programs P1–P11 are relatively small and can be verified in less than 1 second.)

All the experiments on Boolean Programs are conducted under the HW leakage model. We compare the performance of our tool QMVerif with three state-of-the-art tools QMSInfer [44], SC Sniffer [42], [50] and maskVerif [37], which are designed for verifying masking countermeasure of *Boolean programs only*. In particular, SC Sniffer is an SMT-based tool with an incremental heuristic which is similar to our dominated subexpression elimination. Since SC Sniffer is not publicly available, we implemented the algorithms according to the papers [42], [50] for perfect masking verification and computing QMS values. We remark that instead of computing exact QMS values, SC Sniffer approximates QMS values by directly binary searching the QMS value q between 0 to 1 with a pre-defined step size $\epsilon = 0.01$ [50]. Similar to our tool QMVerif, QMSInfer is a tool that integrates a type system and SMT-based model-counting method, and maskVerif is a tool that integrates a proof system and a brute-force enumeration. Note that we do not compare our tool QMVerif with the Fourier analysis based tool rebecca developed by Bloem et al. [67], as rebecca is designed for masked *hardware* Boolean programs and more importantly, maskVerif [37] has turned to be significantly better than rebecca. Since the input format of maskVerif [37] differs from the syntax of P12–P17, we transform P12–P17 into the forms that can be accepted by maskVerif.

5.3.1 Perfect Masking Verification

The experimental results of perfect masking verification on the programs P12–P17 are reported in Table 5. Column 1 shows the name of the program. Column 2 gives the ground truth. Column 3 shows the number of internal variables. Column 4 shows the number of leaky internal variables. Column 5 shows the number of internal variables which needs the model-counting methods. Column 6–7 respectively show the total time of our tool QMVerif using SMT-based and brute-force methods. Columns 8–10 respectively show the total time of the tool QMSInfer [44], the incremental verification method of SC Sniffer [50] and the tool maskVerif [37].

Recall that the syntax of input programs for maskVerif is different from ours, we equivalently transformed the programs P12–P17 into the input syntax of maskVerif. maskVerif arose "Fatal error: exception Stack overflow" on all the Boolean programs during parsing. Therefore, the results of maskVerif in Column 10 are conducted on the reduced programs, where the last 50,000 lines of assignments (out of nearly 210,000 lines of assignments) are removed. Furthermore, on leakage programs P13–P17, maskVerif terminates once a flaw is identified without checking the rest.

The experimental results show that: (1) our tool QMVerif is effective in verifying Boolean programs; and (2) contrary to the results on arithmetic programs, the performance of the SMT-based and brute-force methods in our QMVerif for verifying perfect masking of Boolean programs is largely leveled.

Compared with QMSInfer [44] and SC Sniffer [50] (1) our tool QMVerif is significantly faster (18–213 times) on the leakage programs (i.e., P13–P17); and (2) on the leakage-free program P12, QMVerif is comparable with QMSInfer, but is at least 1,500 times faster than SC Sniffer.

Compared with maskVerif [37], (1) our tool QMVerif is at least 2,000 times faster on the leakage-free program P12; (2) QMVerif also outperforms on the leakage program P13, P14, P16 and P17, although maskVerif terminates immediately once a flaw is identified, while QMVerif identified *all* flaws; and (3) maskVerif failed to verify the leakage program P15 as it contains the bit-wise or operation ($\vee$) which maskVerif does not support. We did not replace the bit-wise or operation ($\vee$) by other bit-wise operations (e.g., and operation ($\wedge$) and negation ($\neg$) operation) supported by maskVerif, as we believe that the experimental results on the Boolean programs considered here suffice to demonstrate the superiority of our tool.

5.3.2 Computing QMS Values

The experimental results of computing QMS values on P13–P17 (P12 is excluded because it does not contain any leaky internal variable) are reported in Table 6. Column 2 shows the number of leaky internal variables. Columns 3–8 show statistics of our tool QMVerif including the number of iterations in binary search (cf. Section 4.4.2), the time of using SMT-based (resp. brute-force) method, the minimal, maximal and average of QMS values. Column 9 shows the time of QMSInfer. Note that QMSInfer gives the same statistics as QMVerif except for time. Columns 10–14 shows the total number of iterations in the binary search, time, the minimal, maximal and average of QMS values using the algorithm from [42]. Note that all the times reported in Table 6 *exclude* the times used for perfect masking verification, and maskVerif [37] does *not* support computing QMS values.

Compared with the two state-of-the-art tools QMSInfer [44] and SC Sniffer [42], our tool QMVerif is significantly faster than them. We mention that the number of iterations in binary search of the tools QMVerif and QMSInfer [44] depends on the number of bits of random variables, while it is fixed in SC Sniffer for each computation. This results in different time performance. In particular, the QMS values of leaky variables whose computations do not contain random variables (e.g., P13 and P17), do not need the binary search. The improvement of QMVerif compared with QMSInfer owns to our heuristics. In terms of accuracy, QMVerif and QMSInfer have the same results, while SC Sniffer sometimes computes approximate QMS values, e.g., P14, P15 and P17. On the other hand, the brute-force method also outperforms the SMT-based method in our tool.

To conclude, our basic findings can be summarized as follows:

- QMVerif is effective to prove security of leakage-free programs and identify flaws of leakage programs, and shows orders of magnitude improvements over the state-of-the-art tools QMSInfer [44], SC Sniffer [42] and maskVerif [37];
- Arithmetic programs (B2A [16], A2B [16], B2A [17], A2B [17] and B2A [19]) can be proved secure automatically computer-aided tools rather than manually;
- The brute-force model-counting method significantly outperforms the SMT-based one on arithmetic programs, and they are roughly comparable on Boolean programs.

6 Related Work

In this section, we discuss masking schemes, verification approaches, mitigation techniques and measurement of information leakage related to power side-channel attacks. Work on other side-channel attacks that rely on execution-time [3], [68], [69], [70], [71], [72], [73], [74], [75], [76], faults [52], [77], [78], [79], and cache [80], [81], [82], [83], [84], [85], [86], [87], [88], [89] do exist, but is orthogonal to ours, hence will not be discussed in this section.

6.1 Masking Schemes

To thwart power side-channel attacks, various masking schemes as countermeasures have been proposed, such as Boolean masking scheme, arithmetic masking scheme and their combination [14], [15], [16], [25], [26], [27], [28], [29],

TABLE 6
Results of computing QMS values on Boolean programs, where column *Name* gives the name of the program, column ♯*Iter* denotes the number of iterations of binary search, column $\sharp\tau_{\mathsf{lk}}$ denotes the number of leaky internal variables, column *SMT* denotes the results of applying the SMT-based method, column *B.F.* denotes the results of applying the brute-force method, columns *Min*, *Max* and *Arg.* respectively give the minimal, maximal and average QMS values, the last six columns respectively show the total verification time of the tools QMSInfer [44] and SC Sniffer [50].

Name	$\sharp\tau_{\mathsf{lk}}$	QMVerif						QMSInfer [44]	SC Sniffer [50]				
		♯Iter	SMT	B.F.	Min	Max	Arg.	Time	♯Iter	Time	Min	Max	Avg.
P13	4.8k	**0**	0	0	0.00	1.00	0.98	0	480k	97m 23s	0.00	1.00	0.98
P14	3.2k	9.6k	2m 56s	**39s**	0.50	1.00	0.99	33m 3s	160k	40m 13s	0.51	1.00	0.99
P15	1.6k	4.8k	1m 36s	**1m 32s**	0.50	1.00	1.00	28m 7s	80k	23m 26s	0.51	1.00	1.00
P16	4.8k	6.4k	1m 40s	**8s**	0.00	1.00	0.98	45m 14s	320k	66m 27s	0.00	1.00	0.98
P17	17.6k	4.8k	51s	**1s**	0.00	1.00	0.94	72m 14s	1440k	337m 46s	0.00	1.00	0.93

[30], [31], [51], [90], [91]. These schemes differ in adversary models, efficiency, cryptographic algorithms and compactness. Countermeasures are often manually designed for specific cryptographic algorithms and implementations of cryptographic algorithms that rely on secure masking schemes are not secure automatically. In this context, there is a shortage of effective and automated tools for proving their security and accurately identifying flaws [32], [33].

6.2 Verification Approaches

We discuss related work on masking countermeasure verification along two categorizations: symbolic approaches and model-counting based approaches.

6.2.1 Symbolic Approaches

Symbolic approaches have been widely used in the verification of side-channel attacks with early work [34], [92], where masking compilers are provided which can transform an input program into a functionally equivalent program that is resistant to first-order DPA. However, these systems either are limited to certain operations (i.e., $\oplus$ and table look-up), or suffer from unsoundness and incompleteness under the threshold probing model [14] or the HW/WD leakage model.

One of the most groundbreaking works in this direction are the works of Barthe et al. [35], [36], [37]. In [35], Barthe et al. introduced the notion of noninterference (NI) and inference system for proving masked arithmetic programs. In [36], Barthe et al. introduced the notion of strong noninterference (SNI) which is an extension of the NI notion. The SNI notion allows to prove the security of masked arithmetic programs compositionally, instead of proving the security of a whole implementation at once. However, their compositional verification requires that the programs are either free of procedure calls, or consist of sequences of procedure calls that satisfy NI/SNI condition and each share is used at most once except for a SNI refresh function. For instance, the procedure *SecExp254* in our running example does not satisfy these conditions. The restriction of NI/SNI procedure calls is addressed in [93], but it is still limited to some specific procedures and Boolean programs only. Recently, Barthe et al. implemented a unified framework for both *Boolean* software and hardware implementations in the tool maskVerif [37], featuring the NI and SNI notions extended of glitches and transitions. Further work along this line includes improvements for efficiency [38], [39] but limited to few certain operations, generalization for assembly-level code [60], [61] and LLVM IR [40], extensions with glitches for hardware programs [94] and extensions with transitions [58], [95].

The HW leakage model considered in this work is equivalent to the first-order threshold probing model [14] and corresponds to the NI notion introduced in [35]. The HD leakage model is equivalent to the first-order threshold probing model [14] and corresponds to the NI notion proposed in [35] with transitions. While the NI/SNI notions [35], [36], [37] are stronger than the HW leakage model, leakage-free programs under the HW leakage model may leak under the NI/SNI notions. It was shown by Wang et al. [40] that the HD leakage model differs from the second-order probing model, hence, also differs from the NI/SNI notions even with glitches. It is unclear whether our approach could be extended to verify programs under the NI/SNI leakage models. Moreover, all the approaches (except [37]) discussed above are *not* complete, i.e., secure programs may fail to pass their verification and spurious flaws cannot be automatically identified, while [37] is limited to Boolean programs *only*. Experimental results on Boolean programs also demonstrate that our tool QMVerif significantly outperforms maskVerif [37].

6.2.2 Model-counting based Approaches

Model-counting based approaches also have been proposed for formally verifying masking countermeasures of cryptographic programs [41], [42], [49], [50], [67], [96], [97]. In [41], Eldib et al. firstly proposed a model-counting based approach by leveraging SMT solvers under the HW leakage model, which is later extended to taking the HD leakage model into account [42] and to quantifying masking strength of resistance using the QMS notion [49], [50] under the HW leakage model. The main advantage of their verification approaches is completeness [41], [42]. However, all the works [41], [42], [49], [50] are limited in scalability and Boolean programs *only*. Blot et al. generalized the SMT-based approach to higher-order Boolean programs and presented compositional rules for fragments of code [97]. However, it requires that all the compositional fragments have disjointed random variables and each sequential composition of two fragments should be connected by a refresh of shares. Another model-counting based approaches solve the verification problem via Fourier analysis [67], [96]. In the nutshell, by using the Fourier expansion of the Boolean functions, they reduce the verification problem under the (higher-order) HW leakage model (or equivalently threshold probing model [14]) with/without glitches to checking whether certain coefficients of the Fourier expansion are zero or not. The latter is solved by leveraging SAT solvers in [67] which is sound but *not* complete. Moreover, [67], [96] are limited to Boolean programs and qualitative analysis under the HW leakage model *only*.

To improve efficiency, a hybrid approach integrating type inference and SMT-based model-counting based approaches was proposed by Zhang et al. [43]. The type system of [43] is inspired by, but goes beyond, the one in [60], [61]. Indeed, the type system from [60], [61] uses syntactic information of the computations, whereas the type system from [43] uses both syntactic and semantic information where the type inference is an iterative process, making use of SMT-based model-counting approach to refine the type dynamically. The hybrid approach is extended to computing exact QMS later [44], but is still limited to Boolean programs and the HW leakage model.

In the preliminary version of this paper [1], we generalize the approach of [43], [44] from the Boolean setting to the arithmetic setting by extending the notation of dominant variables and type inference rules. It not only extends the applicability but also achieves significant improvement in efficiency even for Boolean programs (cf. Table 5). The type system subsumes that of [43], [44], [60], [61] and provides additional inference rules for arithmetic operations; our SMT-based method extends that in [41], [42], [43], [44], [49], [50]; our tool QMVERIF supports both quantitative and qualitative verification of Boolean and arithmetic programs. Moreover, we propose a brute-force method for solving model-counting constraints and additional heuristics, which make our tool more scalable and efficient in practice. Although [60], [61] have already mentioned that solving model-counting via SMT solvers [42], [43] may not be the best approach, we went further by demonstrating that solving model-counting via SMT solvers [42], [43] is doable on Boolean programs and on arithmetic programs without involving (finite-field) multiplication, but may be inferior for arithmetic programs with (finite-field) multiplication. This provides empirical suggestions on which model-counting method should be selected, and suggests potential future research directions of domain specific model-counting methods.

Last but not least, the current work extends the preliminary version [1] on several aspects: (1) it provides a refined narrative of the motivation, a complex running example to better illustrate our techniques, and an extensive literature review together with thorough comparison of the related work; (2) it formulates Algorithm 1 and Algorithm 2 which are only informally described in [1]; (3) it considers the HD leakage model and demonstrates the performance of our approach under the HD leakage model so our approach has broader applicability; (4) it introduces a novel type system supporting for compositional reasoning which significantly improves efficiency; and (5) it conducts considerably more experiments and gives an in-depth analysis of the results.

6.3 Mitigation

Mitigation techniques have been proposed to generate side-channel leakage-free programs. [98] proposed a dual-spacer dual-rail delay-insensitive logic circuit design methodology to mitigate power side-channel attack. It guarantees balanced switching activities between the two rails of each signal, hence makes attackers difficult to compute the correlation of power consumption data. [34], [36], [99], [100] rely on compiler-like pattern matching, the ones proposed in [97], [101], [102] use inductive program synthesis and [40] leverages register allocation and assignment. Some of the work can provide security guarantee which mainly relies on (qualitative) countermeasure verification techniques to find (potential) flaws. In particular, [101] relies upon SMT-based approaches [41], [42], while [36], [40] rely upon sound but incomplete verification approaches. These incomplete approaches may report spurious flaws which could be fixed by post mitigation techniques [36], [40] producing leakage-free programs, but which may incur overhead of the resulting programs. Nevertheless, it would be interesting to investigate whether our new approach can aid in the synthesis of better masking countermeasures, as done in [36], [40], [101].

6.4 Measurement

Quantitative verification of side-channel resistance is related to quantitative information flow (QIF) analysis [103], [104], [105], [106], [107]. QIF measures the flow of information in programs by leveraging notions from information theory, e.g., Shannon entropy and mutual information. The QIF analysis has been investigated for side-channel analysis [108], [109], [110]. There are several key differences between our work and QIF. Firstly, the programs under verification are different. We consider masked programs in straight-line forms, while QIF targets at fully-fledged programs (including branching and loops) so program analysis techniques (e.g., symbolic execution) are needed. Secondly, the metric is different. We use the notion of QMS that is correlated with the number of power traces needed to successfully infer private data, while QIF leverages notions from information theory which is used to quantify the volume of leakages. Finally, although both work rely on model-counting, the constraints in QIF over the input are usually linear, while the ones in our setting involve arithmetic operations in rings and fields. Approximation techniques can be leveraged in QIF [107], [110], but are not suitable for ours. Furthermore, it is worth mentioning that in general input variables in QIF should be partitioned into two disjoint sets (public and private variables), and the former needs to be existentially quantified. This was also observed by, e.g., [110], but without any implementation.

7 Conclusion

In this work, we have proposed an integration of type system and model-counting based methods, aided by heuristics for verifying masking countermeasures of arithmetic programs under both the HW and HD leakage models. The type inference allows an efficient, lightweight procedure to determine most internal variables whereas model-counting accounts for completeness, bringing the best of two worlds. In particular, our type system can support compositional reasoning for programs with procedure calls, which can reduce the need of procedure inlining, and thus substantially improve the efficiency of type inference. We also provided a binary search based algorithm to quantify resistance of masking countermeasures by leveraging model-counting based methods. We have implemented our approach in a verification tool QMVerif and evaluated it on standard cryptographic benchmarks. The experimental results demonstrate that QMVerif is effective to prove security of leakage-free programs and identify flaws of leakage programs in a compositional manner. Furthermore, QMVerif is substantially (order of magnitude in some cases) faster than QMSInfer, SC Sniffer and maskVerif. Several conversion algorithms between Boolean and arithmetic maskings (e.g., B2A [16], A2B [16], B2A [17], A2B [17] and B2A [19]) have been formally proved leakage-free by QMVerif, which were only possible manually in previous work.

Future work includes further investigation of efficient model-counting techniques for domain-specific problems and generalization of the work in the current paper to verification of higher-order masking schemes which remains to be a very challenging task. Under the higher-order setting where the attacker is able to probe multiple variables simultaneously, we have to verify that the joint distribution of each set of the probed variables is statistically independent of the private input variables. There are two technical challenges. First, probed variables may occur in different procedures. In this case, our type system cannot verify each procedure in isolation, calling for a new type system strengthening the compositional reasoning. Second, the number of variables involved in the computation of multiple probed variables may be large, while the complexity of the model-counting based method is exponential in the number of variables. Thus, more efficient model-counting techniques are needed to tackle the scalability.

Another research direction is how to verify programs with inherent branching and loops that cannot be transformed into the straight-line form. Currently, all of the existing security notions (i.e., perfect masking, NI and SNI) are defined over straight-line programs. Whether these notions can be easily adapted to more general programs and how to verify them remain to be an important and challenging problem.

Acknowledgement

The authors would like to thank Professor Gilles Barthe, Professor Benjamin Grégoire and Professor Chao Wang for providing benchmarks, and the anonymous reviewers for their valuable comments and suggestions. P. Gao, H. Xie, P. Sun, J. Zhang and F. Song were partially supported by the National Natural Science Foundation of China (NSFC) grants (No. 61532019 and No. 61761136011); T. Chen was partially supported by UK EPSRC grant (No. EP/P00430X/1), NSFC grant (No. 61872340), Guangdong Science and Technology Department grant (No. 2018B010107004), Natural Science Foundation of Guangdong Province, China (No. 2019A1515011689), and Overseas Grant (KFKT2018A16) from the State Key Laboratory of Novel Software Technology, Nanjing University, China.

References

[1] P. Gao, H. Xie, J. Zhang, F. Song, and T. Chen, "Quantitative verification of masked arithmetic programs against side-channel attacks," in *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2019, pp. 155–173.

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1999, pp. 388–397.

[3] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1996, pp. 104–113.

[4] L. Goubin and J. Patarin, "DES and differential power analysis (the ”duplication” method)," in *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 1999, pp. 158–172.

[5] J. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 1999, pp. 292–302.

[6] C. Clavier, J.-S. Coron, and N. Dabbous, "Differential power analysis in the presence of hardware countermeasures," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 252–263.

[7] K. Itoh, T. Izu, and M. Takenaka, "Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA," in *Proceedings of the4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Revised Papers*, 2002, pp. 129–143.

[8] H. B. Choi, H. J. Lee, C. S. Kim, B. H. Chang, and D. Won, "On differential power analysis attack on the addition modular 2^n operation of smart cards," in *Proceedings of the International Conference on Security and Management (SAM)*, 2003, pp. 260–266.

[9] W. Wang, Y. Yu, F. Standaert, J. Liu, Z. Guo, and D. Gu, "Ridge-based DPA: improvement of differential power analysis for nanoscale chips," *IEEE Trans. Information Forensics and Security*, vol. 13, no. 5, pp. 1301–1316, 2018.

[10] M. J. Kannwischer, A. Genêt, D. Butin, J. Krämer, and J. Buchmann, "Differential power analysis of XMSS and SPHINCS," in *Proceedings of the 9th International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2018, pp. 168–188.

[11] J. Xu, A. Fan, M. Lu, and W. Shan, "Differential power analysis of 8-bit datapath AES for IoT applications," in *Proceedings of the17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering*, 2018, pp. 1470–1473.

[12] C. Luo, Y. Fei, and D. R. Kaeli, "Effective simple-power analysis attacks of elliptic curve cryptography on embedded systems," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018, p. 115.

[13] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[14] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2003, pp. 463–481.

[15] J. Coron and L. Goubin, "On boolean and arithmetic masking against differential power analysis," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2000, pp. 231–237.

[16] L. Goubin, "A sound method for switching between boolean and arithmetic masking," in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, 2001, pp. 3–15.

[17] J. Coron, J. Großschädl, and P. K. Vadnala, "Secure conversion between boolean and arithmetic masking of any order," in *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014, pp. 188–205.

[18] J. Coron, "High-order conversion from boolean to arithmetic masking," in *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017, pp. 93–114.

[19] L. Bettale, J. Coron, and R. Zeitoun, "Improved high-order conversion from boolean to arithmetic masking," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 22–45, 2018.

[20] X. Lai and J. L. Massey, “A proposal for a new block encryption standard,” in *Proceedings of tge Workshop on the Theory and Application of of Cryptographic Techniques (EUROCRYPT)*, 1990, pp. 389–404.

[21] S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin, “Improved analysis of some simplified variants of RC6,” in *Proceedings of the 6th International Workshop on Fast Software Encryption (FSE)*, 1999, pp. 1–15.

[22] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, “The SIMON and SPECK families of lightweight block ciphers,” *IACR Cryptology ePrint Archive*, vol. 2013, p. 404, 2013.

[23] M. Hutter and M. Tunstall, “Constant-time higher-order boolean-to-arithmetic masking,” *J. Cryptographic Engineering*, vol. 9, no. 2, pp. 173–184, 2019.

[24] W. Wang, Y. Yu, and F. Standaert, “Provable order amplification for code-based masking: How to avoid non-linear leakages due to masked operations,” *IEEE Trans. Information Forensics and Security*, vol. 14, no. 11, pp. 3069–3082, 2019.

[25] W. Wang, F. Standaert, Y. Yu, S. Pu, J. Liu, Z. Guo, and D. Gu, “Inner product masking for bitslice ciphers and security order amplification for linear leakages,” in *Proceedings of the 15th International Conference on Smart Card Research and Advanced Applications (CARDIS), Revised Selected Papers*, 2016, pp. 174–191.

[26] T. S. Messerges, “Securing the AES finalists against power analysis attacks,” in *Proceedings of the International Workshop on Fast Software Encryption*, 2000, pp. 150–164.

[27] K. Schramm and C. Paar, “Higher order masking of the AES,” in *Proceedings of the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006, pp. 208–225.

[28] M. Rivain and E. Prouff, “Provably secure higher-order masking of AES,” in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, 2010, pp. 413–427.

[29] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, “Pushing the limits: A very compact and a threshold implementation of AES,” in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2011, pp. 69–88.

[30] E. Prouff and M. Rivain, “Masking against side-channel attacks: A formal security proof,” in *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology*, 2013, pp. 142–159.

[31] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, “Consolidating masking schemes,” in *Annual Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2015, pp. 764–783.

[32] J. Coron, E. Prouff, and M. Rivain, “Side channel cryptanalysis of a higher order masking scheme,” in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, 2007, pp. 28–44.

[33] J. Coron, E. Prouff, M. Rivain, and T. Roche, “Higher-order side channel security and mask refreshing,” in *Proceedings of the International Workshop on Fast Software Encryption*, 2013, pp. 410–424.

[34] A. Moss, E. Oswald, D. Page, and M. Tunstall, “Compiler assisted masking,” in *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2012, pp. 58–75.

[35] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub, “Verified proofs of higher-order masking,” in *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic (EUROCRYPT)*, 2015, pp. 457–485.

[36] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini, “Strong non-interference and type-directed higher-order masking,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 116–129.

[37] G. Barthe, S. Belaïd, P. Fouque, and B. Grégoire, “maskverif: Automated verification of higher-order masking in presence of physical defaults,” in *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, 2019.

[38] E. Bisi, F. Melzani, and V. Zaccaria, “Symbolic analysis of higher-order side channel countermeasures,” *IEEE Trans. Computers*, vol. 66, no. 6, pp. 1099–1105, 2017.

[39] J. Coron, “Formal verification of side-channel countermeasures via elementary circuit transformations,” in *Proceedings of the 16th International Conference on Applied Cryptography and Network Security*, 2018, pp. 65–82.

[40] J. Wang, C. Sung, and C. Wang, “Mitigating power side channels during compilation,” in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[41] H. Eldib, C. Wang, and P. Schaumont, “SMT-based verification of software countermeasures against side-channel attacks,” in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2014, pp. 62–77.

[42] ——, “Formal verification of software countermeasures against side-channel attacks,” *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, p. 11, 2014.

[43] J. Zhang, P. Gao, F. Song, and C. Wang, “SCInfer: Refinement-based verification of software countermeasures against side-channel attacks,” in *Proceedings of the 30th International Conference on Computer Aided Verification, Held as Part of the Federated Logic Conference*, 2018, pp. 157–177.

[44] P. Gao, J. Zhang, F. Song, and C. Wang, “Verifying and quantifying side-channel resistance of masked software implementations,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 16:1–16:32, Jul. 2019.

[45] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-50497-0

[46] H. Groß, D. Schaffenrath, and S. Mangard, “Higher-order side-channel protected implementations of KECCAK,” in *Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 205–212.

[47] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008, pp. 337–340.

[48] M. Nassar, Y. Souissi, S. Guilley, and J. Danger, “RSM: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas,” in *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition*, 2012, pp. 1173–1178.

[49] H. Eldib, C. Wang, M. Taha, and P. Schaumont, “QMS: Evaluating the side-channel resistance of masked software from source code,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2014, pp. 209:1–6.

[50] H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont, “Quantitative masking strength: Quantifying the power side-channel resistance of software code,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1558–1568, 2015.

[51] J. Blömer, J. Guajardo, and V. Krummel, “Provably secure masking of aes,” in *Proceedings of the International Workshop on Selected Areas in Cryptography*. Springer, 2004, pp. 69–83.

[52] H. Eldib, M. Wu, and C. Wang, “Synthesis of fault-attack countermeasures for cryptographic circuits,” in *Proceedings of the International Conference on Computer Aided Verification*, 2016, pp. 343–363.

[53] T. S. Messerges, “Using second-order power analysis to attack dpa resistant software,” in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 238–251.

[54] E. Brier, C. Clavier, and F. Olivier, “Correlation power analysis with a leakage model,” in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2004, pp. 16–29.

[55] A. Moradi, “Side-channel leakage through static power,” in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 562–579.

[56] S. Mangard, “A simple power-analysis (SPA) attack on implementations of the AES key expansion,” in *Proceedings of the 5th International Conference on Information Security and Cryptology*, 2002, pp. 343–358.

[57] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, “Introduction to differential power analysis,” *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[58] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, “On the cost of lazy engineering for masked software implementations,” in *Proceedings of the International Conference on Smart Card Research and Advanced Applications (CARDIS)*, 2014, pp. 64–81.

[59] C. B. Jones, “Specification and design of (parallel) programs,” in *Proceedings of the IFIP 9th World Computer Congress*, 1983, pp. 321–332.

[60] I. B. E. Ouahma, Q. Meunier, K. Heydemann, and E. Encrenaz, “Symbolic approach for side-channel resistance analysis of masked assembly codes,” in *Security Proofs for Embedded Systems*, 2017.

[61] I. B. E. Ouahma, Q. L. Meunier, K. Heydemann, and E. Encrenaz, "Side-channel robustness analysis of masked assembly codes using a symbolic approach," *Journal of Cryptographic Engineering*, vol. 9, no. 3, pp. 231–242, 2019.

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

[63] C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain, "Higher-order masking schemes for s-boxes," in *Proceedings of the 19th International Workshop on Fast Software Encryption. Revised Selected Papers*, 2012, pp. 366–384.

[64] J. Coron, A. Roy, and S. Vivek, "Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Springer, 2014, pp. 170–187.

[65] J. Coron, F. Rondepierre, and R. Zeitoun, "High order masking of look-up tables with common shares," *IACR Cryptology ePrint Archive*, p. 271, 2017.

[66] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2014.

[67] R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter, "Formal verification of masked hardware implementations in the presence of glitches," in *Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology*, 2018, pp. 321–353.

[68] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security Symposium*, 2016, pp. 53–70.

[69] C. S. Pasareanu, Q. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *IEEE Computer Security Foundations Symposium*, 2016, pp. 387–400.

[70] L. Bang, A. Aydin, Q. Phan, C. S. Pasareanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016, pp. 193–204.

[71] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *Proceedings of IEEE Computer Security Foundations Symposium*, 2017, pp. 328–342.

[72] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 362–375.

[73] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 875–890.

[74] T. Brennan, S. Saha, and T. Bultan, "Symbolic path cost analysis for side-channel detection," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 424–425.

[75] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

[76] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, pp. 802–815.

[77] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1997, pp. 513–525.

[78] G. Barthe, F. Dupressoir, P. Fouque, B. Grégoire, and J. Zapalowicz, "Synthesis of fault attacks on cryptographic implementations," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1016–1027.

[79] J. Breier, X. Hou, and Y. Liu, "Fault attacks made easy: Differential fault analysis automation on assembly code," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 96–122, 2018.

[80] P. Grabher, J. Großschädl, and D. Page, "Cryptographic side-channels from low-power cache memory," in *Proceedings of the IMA International Conference, Cirencester on Cryptography and Coding*, 2007, pp. 170–184.

[81] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proceedings of the International Conference on Computer Aided Verification*, 2012, pp. 564–580.

[82] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *USENIX Security Symposium*, 2013, pp. 431–446.

[83] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa, "Leakage resilience against concurrent cache attacks," in *Proceedings of the 3rd International Conference on Principles of Security and Trust, Held as Part of the European Joint Conferences on Theory and Practice of Software*, 2014, pp. 140–158.

[84] D. Chu, J. Jaffar, and R. Maghareh, "Precise cache timing analysis via symbolic execution," in *Proceedings of the IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2016, pp. 293–304.

[85] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leakage in cache attacks via symbolic execution," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 1, pp. 7:1–7:27, 2019.

[86] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 235–252.

[87] C. Sung, B. Paulsen, and C. Wang, "CANAL: A cache timing analysis framework via llvm transformation," in *Proceedings of the IEEE/ACM International Conference On Automated Software Engineering*, 2018.

[88] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2018.

[89] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2018.

[90] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen, "A side-channel analysis resistant description of the AES s-box," in *Proceedings of the International Workshop on Fast Software Encryption*, 2005, pp. 413–423.

[91] D. Canright and L. Batina, "A very compact "perfectly masked" s-box for AES," in *Proceedings of the International Conference on Applied Cryptography and Network Security*, 2008, pp. 446–459.

[92] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, 2013, pp. 293–310.

[93] S. Belaïd, D. Goudarzi, and M. Rivain, "Tight private circuits: Achieving probing security with the least refreshing," in *Proceedings of the 24th International Conferenceon the Theory and Application of Cryptology and Information Security*, 2018, pp. 343–372.

[94] S. Faust, V. Grosso, S. M. D. Pozo, C. Paglialonga, and F. Standaert, "Composable masking schemes in the presence of physical defaults and the robust probing model," *IACR Cryptology ePrint Archive*, vol. 2017, p. 711, 2017.

[95] J. Coron, C. Giraud, E. Prouff, S. Renner, M. Rivain, and P. K. Vadnala, "Conversion of security proofs from one leakage model to another: A new issue," in *Proceedings of the Third International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2012, pp. 69–81.

[96] S. Bhasin, C. Carlet, and S. Guilley, "Theory of masking with codewords in hardware: low-weight dth-order correlation-immune boolean functions," *IACR Cryptology ePrint Archive*, vol. 2013, p. 303, 2013.

[97] A. Blot, M. Yamamoto, and T. Terauchi, "Compositional synthesis of leakage resilient programs," in *Proceedings of the International Conference on Principles of Security and Trust*, 2017, pp. 277–297.

[98] W. Cilio, M. Linder, C. Porter, J. Di, D. R. Thompson, and S. C. Smith, "Mitigating power-and timing-based side-channel attacks using dual-spacer dual-rail delay-insensitive asynchronous logic," *Microelectronics Journal*, vol. 44, no. 3, pp. 258–269, 2013.

[99] A. G. Bayrak, F. Regazzoni, P. Brisk, F. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2011, pp. 230–235.

[100] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2012, pp. 77–82.

[101] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *Proceedings of the International Conference on Computer Aided Verification*, 2014, pp. 114–130.

[102] C. Wang and P. Schaumont, "Security by compilation: an automated approach to comprehensive side-channel resistance," *SIGLOG News*, vol. 4, no. 2, pp. 76–89, 2017.

[103] P. Malacaria and J. Heusser, "Information theory and security: Quantitative information flow," in *Proceedings of the 10th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, 2010, pp. 87–134.

[104] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim, "Quantifying information leaks using reliability analysis," in *Proceedings of 2014 International Symposium on Model Checking of Software (SPIN)*, 2014, pp. 105–108.

[105] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu, "Precisely measuring quantitative information flow: 10k lines of code and beyond," in *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 31–46.

[106] Q. Phan and P. Malacaria, "Abstract model counting: a novel approach for quantification of information leaks," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014, pp. 283–292.

[107] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, "Scalable approximation of quantitative information flow in programs," in *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2018, pp. 71–93.

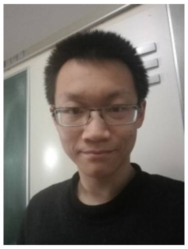
[108] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*, 2017, pp. 328–342.

[109] C. S. Pasareanu, Q. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and Max-SMT," in *Proceedings of the IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 387–400.

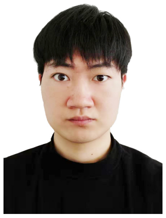
[110] P. Malacaria, M. H. R. Khouzani, C. S. Pasareanu, Q. Phan, and K. S. Luckow, "Symbolic side-channel analysis for probabilistic programs," in *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*, 2018, pp. 313–327.

Pengfei Gao received the B.S. degree in Computer Science from China University of Mining and Technology, Jiangsu, China, in 2017.

He is currently a Ph.D. student in ShanghaiTech University, Shanghai, China, supervised by Prof. Fu Song. His research interests include program analysis and software security.



Hongyi Xie is currently a forth-year undergraduate student in ShanghaiTech University, Shanghai, China, supervised by Prof. Fu Song. His research interests include satisfiability modulo theories and solving of model-counting constraints.



Pu Sun received the B.S. degree in Computer Science from Northeastern University at Qinhuangdao, Hebei, China, in 2018.

He is currently a M.S. student in ShanghaiTech University, Shanghai, China, supervised by Prof. Fu Song. His research interests include software testing and software security.

Jun Zhang received the B.S. degree in Communication Engineering from Shandong University, Shandong, China, in 2016.

He is currently a M.S. student in ShanghaiTech University, Shanghai, China, supervised by Prof. Fu Song. His research interests include program analysis and software security.

Fu Song received the B.S. degree from Ningbo University, Ningbo, China, in 2006, the M.S. degree in Software Engineering from East China Normal University, Shanghai, China, in 2009, and the Ph.D. degree in Computer Science from University Paris-Diderot, Paris, France, in 2013.

From 2013 to 2016, he was a Lecturer and Associate Research Professor at East China Normal University. Since August 2016, he is an Assistant Professor with ShanghaiTech University, Shanghai, China. His research interests include software engineering, formal methods and computer security, especially about automata, logic, model checking, and program analysis.

Dr. Song was a recipient of EASST best paper award at ETAPS 2012.

Taolue Chen received the Bachelor and Master degrees from the Nanjing University, China, both in Computer Science. He was a junior researcher at the Centrum Wiskunde & Informatica (CWI) and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a senior lecturer at the Department of Computer Science, University of Surrey. His research interests include formal verification and synthesis, program analysis, software security, software engineering and machine learning.