Quantitative Verification of Masked Arithmetic Programs against Side-Channel Attacks*

Pengfei Gao[1], Hongyi Xie[1], Jun Zhang[1], Fu Song[1]✉, Taolue Chen[2]

[1] School of Information Science and Technology,
ShanghaiTech University, China
[2] Department of Computer Science and Information Systems,
Birkbeck, University of London, UK

Abstract. Power side-channel attacks, which can deduce secret data via statistical analysis, have become a serious threat. Masking is an effective countermeasure for reducing the statistical dependence between secret data and side-channel information. However, designing masking algorithms is an error-prone process. In this paper, we propose a hybrid approach combing type inference and model-counting to verify masked arithmetic programs against side-channel attacks. The type inference allows an efficient, lightweight procedure to determine most observable variables whereas model-counting accounts for completeness. In case that the program is not perfectly masked, we also provide a method to quantify the security level of the program. We implement our methods in a tool QMVerif and evaluate it on cryptographic benchmarks. The experiment results show the effectiveness and efficiency of our approach.

1 Introduction

Side-channel attacks aim to infer secret data (e.g. cryptographic keys) by exploiting statistical dependence between secret data and non-functional observations such as execution time [33], power consumption [34], and electromagnetic radiation [46]. They have become a serious threat in application domains such as cyber-physical systems. As a typical example, the power consumption of a device executing the instruction $c = p \oplus k$ usually depends on the secret k, and this can be exploited via *differential power analysis* (DPA) [37] to deduce k.

Masking is one of the most widely-used and effective countermeasure to thwart side-channel attacks. Masking is essentially a randomization technique for reducing the statistical dependence between secret data and side-channel information (e.g. power consumption). For example, using Boolean masking scheme, one can mask the secret data k by applying the exclusive-or ($\oplus$) operation with a random variable r, yielding a masked secret data $k \oplus r$. It can be readily verified that the distribution of $k \oplus r$ is independent of the value of k when r is uniformly distributed. Besides Boolean masking scheme, there are other masking schemes

* This work is supported by NSFC(61532019, 61761136011), EPSRC(EP/P00430X/1), and ARC(DP160101652, DP180100691).

such as additive masking schemes (e.g. $(k + r) \bmod n$) and multiplicative masking schemes (e.g. $(k \times r) \bmod n$). A variety of masking implementations such as AES and its non-linear components (S-boxes) have been published over the years. However, designing effective and efficient masking schemes is still a notoriously difficult task, especially for non-linear functions. This has motivated a large amount of work on verifying whether masked implementations, as either (hardware) circuits or (software) programs, are statistically independent of secret inputs. Typically, masked hardware implementations are modeled as (probabilistic) Boolean programs where all variables range over the Boolean domain (i.e. $\mathbb{GF}(2)$), while masked software implementations, featuring a richer set of operations, require to be modeled as (probabilistic) arithmetic programs.

Verification techniques for masking schemes can be roughly classified into type system based approaches [38,3,4,5,16,14,19] and model-counting based approaches [25,24,50]. The basic idea of type system based approaches is to infer a *distribution type* for observable variables in the program that are potentially exposed to attackers. From the type information one may be able to show that the program is secure. This class of approaches is generally very efficient mainly because of their static analysis nature. However, they may give inconclusive answers as most existing type systems do not provide completeness guarantees.

Model-counting based approaches, unsurprisingly, encode the verification problem as a series of model-counting problems, and typically leverage SAT/SMT solvers. The main advantage of this approach is its completeness guarantees. However, the size of the model-counting constraint is usually exponential in the number of (bits of) random variables used in masking, hence the approach poses great challenges to its scalability. We mention that, within this category, some work further exploits Fourier analysis [11,15], which considers the Fourier expansion of the Boolean functions. The verification problem can then be reduced to checking whether certain coefficients of the Fourier expansion are zero or not. Although there is no hurdle in principle, to our best knowledge currently model-counting based approaches are limited to Boolean programs only.

While verification of masking for Boolean programs is well-studied [50,24], generalizing them to arithmetic programs brings additional challenges. First of all, arithmetic programs admit more operations which are absent from Boolean programs. A typical example is field multiplication. In the Boolean domain, it is nothing more than $\oplus$ which is a bit operation. However for $\mathbb{GF}(2^n)$ (typically $n = 8$ in cryptographic algorithm implementations), the operation is nontrivial which prohibits many optimization which would otherwise be useful for Boolean domains. Second, verification of arithmetic programs often suffers from serious scalability issues, especially when the model-counting based approaches are applied. We note that transforming arithmetic programs into equivalent Boolean versions is theoretically possible, but suffer from several deficiencies: (1) one has to encode complicated arithmetic operations (e.g. finite field multiplication) as bitwise operations; (2) the resulting Boolean program needs to be checked against higher-order attacks which are supposed to observe multiple observa-

tions simultaneously. This is a far more difficult problem. Because of this, we believe such as approach is practically, if not infeasible, unfavourable.

Perfect masking is ideal but not necessarily holds when there are flaws or only a limited number of random variables are allowed for efficiency consideration. In case that the program is not perfectly masked (i.e., a potential side channel does exist), naturally one wants to tell how severe it is. For instance, one possible measure is the resource the attacker needs to invest in order to infer the secret from the side channel. For this purpose, we adapt the notion of *Quantitative Masking Strength*, with which a correlation of the number of power traces to successfully infer secret has been established empirically [26,27].

Main contributions. We mainly focus on the verification of masked *arithmetic* programs. We advocate a hybrid verification method combining type system based and model-counting based approaches, and provide additional quantitative analysis. We summarize the main contributions as follows.

- We provide a hybrid approach which integrates type system based and model-counting based approaches into a framework, and support a sound and complete reasoning of masked arithmetic programs.
- We provide quantitative analysis in case when the masking is not effective, to calculate a quantitative measure of the information leakage.
- We provide various heuristics and optimized algorithms to significantly improve the scalability of previous approaches.
- We implement our approaches in a software tool and provide thorough evaluations. Our experiments show orders of magnitude of improvement with respect to previous verification methods on common benchmarks.

We find, perhaps surprisingly, that for model-counting, the widely adopted approaches based on SMT solvers (e.g. [25,24,50]) may not be the best approach, as our experiments suggest that an alternative brute-force approach is comparable for Boolean programs, and significantly outperforms for arithmetic programs.

Related work. The d-threshold probing model is the de facto standard leakage model for order-d power side-channel attacks [32]. This paper focuses on the case that $d = 1$. Other models like noise leakage model [17,45], bounded moment model [6], and threshold probing model with transitions/glitch [20,15] could be reduced to the threshold probing model, at the cost of introducing higher orders [3]. Other work on side channels such as execution-time, faults, and cache do exist ([33,1,2,12,28,7,8,31] to cite a few), but is orthogonal to our work.

Type systems have been widely used in the verification of side channel attacks with early work [38,9], 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. To support verification of higher-order masking, Barthe et al. introduced the notion of noninterference (NI, [3]), and strong t-noninterference (SNI, [4]), which were extended to give a unified framework for both software and hardware implementations in `maskVerif` [5].

Further work along this line includes improvements for efficiency [14,19], generalization for assembly-level code [15], and extensions with glitches for hardware programs [29]. As mentioned earlier, these approaches are incomplete, i.e., secure programs may fail to pass their verification.

[25,24] proposed a model-counting based approach for Boolean programs by leveraging SMT solvers, which is complete but limited in scalability. To improve efficiency, a hybrid approach integrating type-based and model-counting based approaches [25,24] was proposed in [50], which is similar to the current work in spirit. However, it is limited to Boolean programs and qualitative analysis only. [26,27] extended the approach of [25,24] for quantitative analysis, but is limited to Boolean programs. The current work not only extends the applicability but also achieves significant improvement in efficiency even for Boolean programs (cf. Section 5). We also find that solving model-counting via SMT solvers [24,50] may not be the best approach, in particular for arithmetic programs.

Our work is related to quantitative information flow (QIF) [35,44,49,43,13] which leverages notions from information theory (typically Shannon entropy and mutual information) to measure the flow of information in programs. The QIF framework has also been specialized to side-channel analysis [42,41,36]. The main differences are, first of all, QIF targets fully-fledged programs (including branching and loops) so program analysis techniques (e.g. symbolic execution) are needed, while we deal with more specialized (transformed) masked programs in straight-line forms; second, to measure the information leakage quantitatively, our measure is based on the notion QMS which is correlated with the number of power traces needed to successfully infer the secret, while QIF is based on a more general sense of information theory; third, for calculating such a measure, both works rely on model-counting. In QIF, the constraints over the input are usually linear, but the constraints in our setting involve arithmetic operations in rings and fields. Randomized approximate schemes can be exploited in QIF [13] which is not suitable in our setting. Moreover, we mention that in QIF, input variables should in principle be partitioned into public and private variables, and the former of which needs to be existentially quantified. This was briefly mentioned in, e.g., [36], but without implementation.

2 Preliminaries

Let us fix a bounded integer domain $\mathbb{D} = \{0, \cdots, 2^n - 1\}$, where n is a fixed positive integer. Bit-wise operations are defined over $\mathbb{D}$, but we shall also consider arithmetic operations over $\mathbb{D}$ which include $+, -, \times$ modulo 2^n for which $\mathbb{D}$ is consider to be a ring and the Galois field multiplication $\odot$ where $\mathbb{D}$ is isomorphic to $\mathbb{GF}(2)[x]/(p(x))$ (or simply $\mathbb{GF}(2^n)$) for some irreducible polynomial p. For instance, in AES one normally uses $\mathbb{GF}(2^8)$ and $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

2.1 Cryptographic Programs

We focus on programs written in C-like code that implement cryptographic algorithms such as AES, as opposed to arbitrary software programs. To analyze such

programs, it is common to assume that they are given in straight-line forms (i.e., branching-free) over $\mathbb{D}$ [24,3]. The syntax of the program under consideration is given as follows, where $c \in \mathbb{D}$.

$$\begin{array}{lr}
\text{Operation:} & \mathcal{O} \ni \circ ::= \oplus \mid \wedge \mid \vee \mid \odot \mid + \mid - \mid \times \\
\text{Expression:} & e ::= c \mid x \mid e \circ e \mid \neg e \mid e \ll c \mid e \gg c \\
\text{Statememt:} & \texttt{stmt} ::= x \leftarrow e \mid \texttt{stmt}; \texttt{stmt} \\
\text{Program:} & P(X_p, X_k, X_r) ::= \texttt{stmt}; \texttt{return}\ x_1, ..., x_m
\end{array}$$

A program P consists of a sequence of assignments followed by a return statement. An assignment $x \leftarrow e$ assigns the value of the expression e to the variable x, where e is built up from a set of variables and constants using (1) bit-wise operations *negation* ($\neg$), *and* ($\wedge$), *or* ($\vee$), *exclusive-or* ($\oplus$), *left shift* $\ll$ and *right shift* $\gg$; (2) modulo 2^n arithmetic operations: *addition* ($+$), *subtraction* ($-$), *multiplication* ($\times$); and (3) finite-field *multiplication* ($\odot$) (over $\mathbb{GF}(2^n)$)[3]. We denote by $\mathcal{O}^*$ the extended set $\mathcal{O} \cup \{\ll, \gg\}$ of operations.

Given a program P, let $X = X_p \uplus X_k \uplus X_i \uplus X_r$ denote the set of variables used in P, where X_p, X_k and X_i respectively denote the set of public input, private input and internal variables, and X_r denotes the set of (uniformly distributed) random variables for *masking* private variables. We assume that the program is given in the *single static assignment* (SSA) form (i.e., each variable is defined exactly once) and each expression uses at most one operator. (One can easily transform an arbitrary straight-line program into an equivalent one satisfying these conditions.) For each assignment $x \leftarrow e$ in P, **the computation $\mathcal{E}(x)$ of** x is an expression obtained from e by iteratively replacing all the occurrences of the internal variables in e by their defining expressions in P. SSA form guarantees that $\mathcal{E}(x)$ is well-defined.

Semantics. A *valuation* is a function $\sigma : X_p \cup X_k \to \mathbb{D}$ assigning to each variable $x \in X_p \cup X_k$ a value $c \in \mathbb{D}$. Let Θ denote the set of all valuations. Two valuations $\sigma_1, \sigma_2 \in \Theta$ are Y*-equivalent*, denoted by $\sigma_1 \approx_Y \sigma_2$, if $\sigma_1(x) = \sigma_2(x)$ for all $x \in Y$.

Given an expression e in terms of $X_p \cup X_k \cup X_r$ and a valuation $\sigma \in \Theta$, we denote by $e(\sigma)$ the expression obtained from e by replacing all the occurrences of variables $x \in X_p \cup X_k$ by their values $\sigma(x)$, and denote by $[\![e]\!]_\sigma$ the distribution of e (with respect to the uniform distribution of random variables $e(\sigma)$ may contain). Concretely, $[\![e]\!]_\sigma(v)$ is the probability of the expression $e(\sigma)$ being evaluated to v for each $v \in \mathbb{D}$. For each variable $x \in X$ and valuation $\sigma \in \Theta$, we denote by $[\![x]\!]_\sigma$ the distribution $[\![\mathcal{E}(x)]\!]_\sigma$. The semantics of the program P is defined as a (partial) function $[\![P]\!]$ which takes a valuation $\sigma \in \Theta$ and an internal variable $x \in X_i$ as inputs, returns the distribution $[\![x]\!]_\sigma$ of x.

Threat models and security notions. We assume that the adversary has access to public input X_p, but not to private input X_k or random variables X_r, of a program P. However, the adversary may have access to an internal variable $x \in X_i$ via side-channels. Under these assumptions, the goal of the adversary is to deduce the information of X_k.

[3] Note that addition/subtraction over Galois fields is essentially bit-wise exclusive-or.

```
1 Cube(k, r0, r1){
2     x = k ⊕ r0 ;
3     x0 = x ⊙ x ;
4     x1 = r0 ⊙ r0 ;
5     x2 = x0 ⊙ r0 ;
6     x3 = x1 ⊙ x ;
7     x4 = r1 ⊕ x2 ;
8     x5 = x4 ⊕ x3 ;
9     x6 = x0 ⊙ x ;
10    x7 = x6 ⊕ r1 ;
11    x8 = x1 ⊙ r0 ;
12    x9 = x8 ⊕ x5 ;
13    return (x7, x9) ;
14 }
```

Fig. 1. A buggy version of Cube from [47]

Definition 1. *Let P be a program. For every internal variable $x \in X_i$,*

- *x is* uniform *in P, denoted by x-**UF**, if $[\![P]\!](\sigma)(x)$ is uniform for all $\sigma \in \Theta$.*
- *x is* statistically independent *in P, denoted by x-**SI**, if $[\![P]\!](\sigma_1)(x) = [\![P]\!](\sigma_2)(x)$ for all $(\sigma_1, \sigma_2) \in \Theta^2_{X_p}$, where $\Theta^2_{X_p} := \{(\sigma_1, \sigma_2) \in \Theta \times \Theta \mid \sigma_1 \approx_{X_p} \sigma_2\}$.*

Proposition 1. *If the program P is x-**UF**, then P is x-**SI**.*

Definition 2. *For a program P, a variable x is* perfectly masked *(a.k.a. secure under 1-threshold probing model [32]) in P if it is x-**SI**, otherwise x is leaky.*

P is perfectly masked *if all internal variables in P are perfectly masked.*

2.2 Quantitative Masking Strength

When a program is not perfectly masked, it is important to quantify how secure it is. For this purpose, we adapt the notion of *Quantitative Masking Strength* (QMS) from [26,27] to quantify the strength of masking countermeasures.

Definition 3. *The* quantitative masking strength $\mathtt{QMS}_x$ *of a variable $x \in X$, is defined as:* $1 - \max_{(\sigma_1,\sigma_2)\in\Theta^2_{X_p}, c\in\mathbb{D}} \left([\![x]\!]_{\sigma_1}(c) - [\![x]\!]_{\sigma_2}(c) \right)$.

Accordingly, the quantitative masking strength of the program P is defined by $\mathtt{QMS}_P := \min_{x\in X_i} \mathtt{QMS}_x$.

The notion of QMS generalizes that of perfect masking, i.e., P is x-**SI** iff $\mathtt{QMS}_x = 1$. The importance of QMS has been highlighted in [26,27] where it is empirically shown that, for Boolean programs the number of power traces needed to determine the secret key is exponential in the QMS value. This study suggests that computing an *accurate* QMS value for leaky variables is highly desirable.

Example 1. Let us consider the program in Fig. 1, which implements a buggy Cube in $\mathbb{GF}(2^8)$ from [47]. Given a secret key k, to avoid first-order side-channel attacks, k is masked by a random variable r_0 leading to two shares $x = k \oplus r_0$ and r_0. $\mathtt{Cube}(k, r_0, r_1)$ returns two shares x_7 and x_9 such that $x_7 \oplus x_9 = k^3 := k \odot k \odot k$, where r_1 is another random variable.

`Cube` computes $k \odot k$ by $x_0 = x \odot x$ and $x_1 = r_0 \odot r_0$ (Lines 3-4), as $k \odot k = x_0 \oplus x_1$. Then, it computes k^3 by a secure multiplication of two pairs of shares (x_0, x_1) and (x, r_0) using the random variable r_1 (Lines 5-12). However, this program is vulnerable to first-order side-channel attacks, as it is neither x_2-**SI** nor x_3-**SI**. As shown in [47], we shall refresh (x_0, x_1) before computing $k^2 \odot k$ by

inserting $x_0 = x_0 \oplus r_2$ and $x_1 = x_1 \oplus r_2$ after Line 4, where r_2 is a random variable. We use this buggy version as a running example to illustrate our techniques.

As setup for further use, we have: $X_p = \emptyset$, $X_k = \{k\}$, $X_r = \{r_0, r_1\}$ and $X_i = \{x, x_0, \cdots, x_9\}$. The computations $\mathcal{E}(\cdot)$ of internal variables are:

$\mathcal{E}(x) = k \oplus r_0$ $\quad \mathcal{E}(x_0) = (k \oplus r_0) \odot (k \oplus r_0)$ $\quad \mathcal{E}(x_1) = r_0 \odot r_0$

$\mathcal{E}(x_2) = ((k \oplus r_0) \odot (k \oplus r_0)) \odot r_0$ $\quad \mathcal{E}(x_3) = (r_0 \odot r_0) \odot (k \oplus r_0)$

$\mathcal{E}(x_4) = r_1 \oplus (((k \oplus r_0) \odot (k \oplus r_0)) \odot r_0)$ $\quad \mathcal{E}(x_6) = ((k \oplus r_0) \odot (k \oplus r_0)) \odot (k \oplus r_0)$

$\mathcal{E}(x_5) = (r_1 \oplus ((k \oplus r_0) \odot (k \oplus r_0)) \odot r_0) \oplus ((r_0 \odot r_0) \odot (k \oplus r_0))$

$\mathcal{E}(x_7) = (((k \oplus r_0) \odot (k \oplus r_0)) \odot (k \oplus r_0)) \oplus r_1$ $\quad \mathcal{E}(x_8) = (r_0 \odot r_0) \odot r_0$

$\mathcal{E}(x_9) = ((r_0 \odot r_0) \odot r_0) \oplus ((r_1 \oplus ((k \oplus r_0) \odot (k \oplus r_0) \odot r_0)) \oplus ((r_0 \odot r_0) \odot (k \oplus r_0)))$

3 Three Key Techniques

In this section, we introduce three key techniques: type system, model-counting based reasoning and reduction techniques, which will be used in our algorithm.

3.1 Type System

We present a type system for formally inferring *distribution types* of internal variables, inspired by prior work [40,3,14,50]. We start with some basic notations.

Definition 4 (Dominant variables). *Given an expression e, a random variable r is called a* dominant variable *of e if the following two conditions hold: (i) r occurs in e exactly once, and (ii) each operator on the path between the leaf r and the root in the abstract syntax tree of e is from either $\{\oplus, \neg, +, -\}$ or $\{\times, \odot\}$ such that one of the children of the operator is a non-zero constant.*

Remark that in Definition 4, for efficiency consideration, we take a purely syntactic approach meaning that we do not simplify e when checking the condition (i) that r occurs once. For instance, x is *not* a dominant variable in $((x \oplus y) \oplus x) \oplus x$, although intuitively e is equivalent to $y \oplus x$.

Given an expression e, let $\mathsf{Var}(e)$ be the set of variables occurring in e, and $\mathsf{RVar}(e) := \mathsf{Var}(e) \cap X_r$. We denote by $\mathsf{Dom}(e) \subseteq \mathsf{RVar}(e)$ the set of all dominant random variables of e, which can be computed in linear time in the size of e.

Proposition 2. *Given a program P with $\mathcal{E}(x)$ defined for each variable x of P, if $\mathsf{Dom}(\mathcal{E}(x)) \neq \emptyset$, then P is x-**UF**.*

Definition 5 (Distribution Types). *Let $\mathcal{T} = \{\mathsf{RUD}, \mathsf{SID}, \mathsf{SDD}, \mathsf{UKD}\}$ be the set of distribution types, where for each variable $x \in X$,*

- $\mathcal{E}(x) : \mathsf{RUD}$ *meaning that the program is x-**UF**;*
- $\mathcal{E}(x) : \mathsf{SID}$ *meaning that the program is x-**SI**;*
- $\mathcal{E}(x) : \mathsf{SDD}$ *meaning that the program is not x-**SI**;*
- $\mathcal{E}(x) : \mathsf{UKD}$ *meaning that the distribution type of x is unknown.*

where RUD *is a subtype of* SID *(cf. Proposition 1).*

$$\frac{\mathsf{Dom}(e) \neq \emptyset}{\vdash e : \mathsf{RUD}}\ (\textsc{Dom}) \qquad \frac{\vdash e_1 \star e_2 : \tau}{\vdash e_2 \star e_1 : \tau}\ (\textsc{Com}) \qquad \frac{\vdash e : \tau}{\vdash \neg e : \tau}\ (\textsc{Ide}_1)$$

$$\frac{\vdash e : \mathsf{SID}}{\vdash e \bullet e : \mathsf{SID}}\ (\textsc{Ide}_2) \qquad \frac{}{\vdash e \diamond e : \mathsf{SID}}\ (\textsc{Ide}_3) \qquad \frac{\vdash e : \mathsf{SDD}}{\vdash e \bowtie e : \mathsf{SDD}}\ (\textsc{Ide}_4)$$

$$\frac{\mathsf{Var}(e) \cap X_k = \emptyset}{\vdash e : \mathsf{SID}}\ (\textsc{NoKey}) \qquad \frac{x \in X_k}{\vdash x : \mathsf{SDD}}\ (\textsc{Key}) \qquad \frac{\begin{array}{c}\vdash e_1 : \mathsf{RUD} \ \ \vdash e_2 : \mathsf{RUD} \\ \mathsf{Dom}(e_1) \setminus \mathsf{RVar}(e_2) \neq \emptyset\end{array}}{\vdash e_1 \circ e_2 : \mathsf{SID}}\ (\textsc{Sid}_1)$$

$$\frac{\begin{array}{c}\vdash e_1 : \mathsf{SID} \ \ \vdash e_2 : \mathsf{SID} \\ \mathsf{RVar}(e_1) \cap \mathsf{RVar}(e_2) = \emptyset\end{array}}{\vdash e_1 \bullet e_2 : \mathsf{SID}}\ (\textsc{Sid}_2) \qquad \frac{\begin{array}{c}\vdash e_1 : \mathsf{SDD} \ \ \vdash e_2 : \mathsf{RUD} \\ \mathsf{Dom}(e_2) \setminus \mathsf{RVar}(e_1) \neq \emptyset\end{array}}{\vdash e_1 \circ e_2 : \mathsf{SDD}}\ (\textsc{Sdd}) \qquad \frac{\begin{array}{c}\text{No rule is} \\ \text{appliable to } e\end{array}}{\vdash e : \mathsf{UKD}}\ (\textsc{Ukd})$$

Fig. 2. Type inference rules, where $\star \in \mathcal{O}$, $\circ \in \{\wedge, \vee, \odot, \times\}$, $\bullet \in \mathcal{O}^*$, $\bowtie \in \{\wedge, \vee\}$ and $\diamond \in \{\oplus, -\}$.

Type judgements, as usual, are defined in the form of $\vdash e : \tau$, where e is an expression in terms of $X_r \cup X_k \cup X_p$, and $\tau \in \mathcal{T}$ denotes the distribution type of e. A type judgement $\vdash e : \mathsf{RUD}$ (resp. $\vdash e : \mathsf{SID}$ and $\vdash e : \mathsf{SDD}$) is valid iff P is x-**UF** (resp. x-**SI** and not x-**SI**) for all variables x such that $\mathcal{E}(x) = e$. A sound proof system for deriving valid type judgements is given in Figure 2.

Rule (Dom) states that e containing some dominant variable has type RUD (cf. Proposition 2). Rule (Com) captures the commutative law of operators $\star \in \mathcal{O}$. Rules ($\textsc{Ide}_i$) for $i = 1, 2, 3, 4$ are straightforward. Rule (NoKey) states that e has type SID if e does not use any private input. Rule (Key) states that each private input has type SDD. Rule ($\textsc{Sid}_1$) states that $e_1 \circ e_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$ has type SID, if both e_1 and e_2 have type RUD, and e_1 has a dominant variable r which is not used by e_2. Indeed, $e_1 \circ e_2$ can be seen as $r \circ e_2$, then for each valuation $\eta \in \Theta$, the distributions of r and $e_2(\eta)$ are independent. Rule ($\textsc{Sid}_2$) states that $e_1 \bullet e_2$ for $\bullet \in \mathcal{O}^*$ has type SID, if both e_1 and e_2 have type SID (as well as its subtype RUD), and the sets of random variables used by e_1 and e_2 are disjoint. Likewise, for each valuation $\eta \in \Theta$, the distributions on $e_1(\eta)$ and $e_2(\eta)$ are independent. Rule (Sdd) states that $e_1 \circ e_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$ has type SDD, if e_1 has type SDD, e_2 has type RUD, and e_2 has a dominant variable r which is not used by e_1. Intuitively, $e_1 \circ e_2$ can be safely seen as $e_1 \circ r$.

Finally, if no rule is applicable to an expression e, then e has unknown distribution type. Such a type is needed because our type system is—by design—incomplete. However, we expect—and demonstrate empirically—that for cryptographic programs, most internal variables have a definitive type other than UKD. As we will show later, to resolve UKD-typed variables, one can resort to model-counting (cf. Section 3.2).

Theorem 1. *If* $\vdash \mathcal{E}(x) : \mathsf{RUD}$ *(resp.* $\vdash \mathcal{E}(x) : \mathsf{SID}$ *and* $\vdash \mathcal{E}(x) : \mathsf{SDD}$*) is valid, then* P *is* x-**UF** *(resp.* x-**SI** *and not* x-**SI***).*

Example 2. Consider the program in Figure 1, we have:

$$\begin{array}{llll} \vdash \mathcal{E}(x) : \mathsf{RUD}; & \vdash \mathcal{E}(x_0) : \mathsf{SID}; & \vdash \mathcal{E}(x_1) : \mathsf{SID}; & \vdash \mathcal{E}(x_2) : \mathsf{UKD}; \\ \vdash \mathcal{E}(x_3) : \mathsf{UKD}; & \vdash \mathcal{E}(x_4) : \mathsf{RUD}; & \vdash \mathcal{E}(x_5) : \mathsf{RUD}; & \vdash \mathcal{E}(x_6) : \mathsf{UKD}; \\ \vdash \mathcal{E}(x_7) : \mathsf{RUD}; & \vdash \mathcal{E}(x_8) : \mathsf{SID}; & \vdash \mathcal{E}(x_9) : \mathsf{RUD}. & \end{array}$$

3.2 Model-Counting based Reasoning

Recall that for $x \in X_i$, $\mathtt{QMS}_x := 1 - \max_{(\sigma_1,\sigma_2)\in\Theta^2_{X_p}, c\in\mathbb{D}}(\llbracket x\rrbracket_{\sigma_1}(c) - \llbracket x\rrbracket_{\sigma_2}(c))$.

To compute $\mathtt{QMS}_x$, one naive approach is to use brute-force to enumerate all possible valuations σ and then to compute distributions $\llbracket x\rrbracket_\sigma$ again by enumerating the assignments of random variables. This approach is exponential in the number of (bits of) variables in $\mathcal{E}(x)$.

Another approach is to lift the SMT-based approach [26,27] from Boolean setting to the arithmetic one. We first consider a "decision" version of the problem, i.e., checking whether $\mathtt{QMS}_x \geq q$ for a given rational number $q \in [0,1]$. It is not difficult to observe that this can be reduced to checking the satisfiability of the following logic formula:

$$\exists \sigma_1, \sigma_2 \in \Theta^2_{X_p}.\exists c \in \mathbb{D}.\big(\sharp(c = \llbracket x\rrbracket_{\sigma_1}) - \sharp(c = \llbracket x\rrbracket_{\sigma_2})\big) > \Delta^q_x, \tag{1}$$

where $\sharp(c = \llbracket x\rrbracket_{\sigma_1})$ and $\sharp(c = \llbracket x\rrbracket_{\sigma_2})$ respectively denote the number of satisfying assignments of $c = \llbracket x\rrbracket_{\sigma_1}$ and $c = \llbracket x\rrbracket_{\sigma_2}$, $\Delta^q_x = (1-q)\times 2^m$, and m is the number of bits of random variables in $\mathcal{E}(x)$.

We further encode (1) as a (quantifier-free) first-order formula Ψ^q_x to be solved by an off-the-shelf SMT solver (e.g. Z3 [23]):

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

where

- **Program logic** (Θ_f and Θ'_f): for every $f : \mathsf{RVar}(\mathcal{E}(x)) \to \mathbb{D}$, Θ_f encodes $c_f = \mathcal{E}(x)$ into a logical formula with each occurrence of a random variable $r \in \mathsf{RVar}(\mathcal{E}(x))$ being replaced by its value $f(r)$, where c_f is a fresh variable. There are $|\mathbb{D}|^{|\mathsf{RVar}(\mathcal{E}(x))|}$ distinct copies, but share the same X_p and X_k. Θ'_f is similar to Θ_f except that all variables $k \in X_k$ and c_f are replaced by fresh variables k' and c'_f respectively.
- **Boolean to integer** ($\Theta_{\mathtt{b2i}}$ and $\Theta'_{\mathtt{b2i}}$): $\Theta_{\mathtt{b2i}} := \bigwedge_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathbb{D}} I_f = (c = c_f)\ ?\ 1 : 0$. It asserts that for each $f : \mathsf{RVar}(\mathcal{E}(x)) \to \mathbb{D}$, a fresh integer variable I_f is 1 if $c = c_f$, otherwise 0. $\Theta'_{\mathtt{b2i}}$ is similar to $\Theta_{\mathtt{b2i}}$ except that I_f and c_f are replaced by I'_f and c'_f respectively.
- **Different sums** ($\Theta^q_{\mathtt{diff}}$): $\sum_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathbb{D}} I_f - \sum_{f:\mathsf{RVar}(\mathcal{E}(x))\to\mathbb{D}} I'_f > \Delta^q_x$.

Theorem 2. *Ψ^q_x is* unsatisfiable *iff $\mathtt{QMS}_x \geq q$, and the size of Ψ^q_x is polynomial in $|P|$ and exponential in $|\mathsf{RVar}(\mathcal{E}(x))|$ and $|\mathbb{D}|$.*

Based on Theorem 2, we present an algorithm for computing $\mathtt{QMS}_x$ in Section 4.2.

Note that the qualitative variant of Ψ^q_x (i.e. $q = 1$) can be used to decide whether x is statistically independent by checking whether $\mathtt{QMS}_x = 1$ holds. This will be used in Algorithm 1.

Example 3. By applying the model-counting based reasoning to the program in Figure 1, we can conclude that x_6 is perfectly masked, while x_2 and x_3 are leaky. This cannot be done by our type system or the ones in [3,4]. To give a sample encoding, consider the variable x_3 for $q = \frac{1}{2}$ and $\mathbb{D} = \{0,1,2,3\}$. We have that $\Psi^{\frac{1}{2}}_{x_3}$ is

$$\begin{pmatrix} c_0 = (0 \odot 0) \odot (k \oplus 0) & \wedge & c'_0 = (0 \odot 0) \odot (k' \oplus 0) & \wedge \\ c_1 = (1 \odot 1) \odot (k \oplus 1) & \wedge & c'_1 = (1 \odot 1) \odot (k' \oplus 1) & \wedge \\ c_2 = (2 \odot 2) \odot (k \oplus 2) & \wedge & c'_2 = (2 \odot 2) \odot (k' \oplus 2) & \wedge \\ c_3 = (3 \odot 3) \odot (k \oplus 3) & \wedge & c'_3 = (3 \odot 3) \odot (k' \oplus 3) & \wedge \end{pmatrix}$$
$$\begin{pmatrix} I_0 = (c = c_0)\ ?\ 1 : 0 & \wedge & I_1 = (c = c_1)\ ?\ 1 : 0 & \wedge \\ I_2 = (c = c_2)\ ?\ 1 : 0 & \wedge & I_3 = (c = c_3)\ ?\ 1 : 0 & \wedge \end{pmatrix}$$
$$\begin{pmatrix} I'_0 = (c = c'_0)\ ?\ 1 : 0 & \wedge & I'_1 = (c = c'_1)\ ?\ 1 : 0 & \wedge \\ I'_2 = (c = c'_2)\ ?\ 1 : 0 & \wedge & I'_3 = (c = c'_3)\ ?\ 1 : 0 & \wedge \end{pmatrix}$$
$$(I_0 + I_1 + I_2 + I_3) - (I'_0 + I'_1 + I'_2 + I'_3) > (1 - \tfrac{1}{2})^2$$

3.3 Reduction Heuristics

In this section, we provide various heuristics to reduce the size of formulae. These can be both applied to type inference and model-counting based reasoning.

Ineffective variable elimination. A variable x is *ineffective* in an expression e if for all valuations $\sigma_1, \sigma_2 \in \Theta$ such that $\sigma_1 \approx_{\mathsf{Var}(e)\setminus\{x\}} \sigma_2$, we have $\sigma_1(e) = \sigma_2(e)$. Otherwise, we say x is *effective* in e. Clearly if x is ineffective in e, then e and $e[c/x]$ are equivalent for any $c \in \mathbb{D}$ while $e[c/x]$ contains less variables, where $e[c/x]$ is obtained from e by replacing all occurrences of x with c.

Checking whether x is effective or not in e can be performed by a satisfiability checking of the logical formula: $e[c/x] \neq e[c'/x]$. Obviously, $e[c/x] \neq e[c'/x]$ is satisfiable iff x is effective in e.

Algebraic laws. For every sub-expression e' of the form $e_1 \oplus e_1, e_2 - e_2, e \circ 0$ or $0 \circ e$ with $\circ \in \{\times, \odot, \wedge\}$ in the expression e, it is safe to replace e' by 0, namely, e and $e[0/e']$ are equivalent. Note that the constant 0 is usually introduced by instantiating ineffective variables by 0 when eliminating ineffective variables.

Dominated Subexpression Elimination. Given an expression e, if e' is a r-dominated sub-expression in e and r does not occur in e elsewhere, then it is safe to replace each occurrence of e' in e by the random variable r. Intuitively, e' as a whole can be seen as a random variable when evaluating e. Besides this elimination, we also allow to add mete-theorems specifying forms of sub-expressions e' that can be replaced by a fresh variable. For instance, $r \oplus ((2 \times r) \wedge e'')$ in e, when the random variable r does not appear elsewhere, can be replaced by the random variable r.

Let $\widehat{e}$ denote the expression obtained by applying the above heuristics on the expression e.

Transformation Oracle. We suppose there is an oracle Ω which, whenever possible, transforms an expression e into an equivalent expression $\Omega(e)$ such that the type inference can give a non-UKD type to $\Omega(e)$.

Lemma 1. *$\mathcal{E}(x)(\sigma)$ and $\widehat{\mathcal{E}(x)}(\sigma)$ have same distribution for any $\sigma \in \Theta$.*

Example 4. Consider the variable x_6 in the program in Figure 1, $(k \oplus r_0)$ is r_0-dominated sub-expression in $\mathcal{E}(x_6) = ((k \oplus r_0) \odot (k \oplus r_0)) \odot (k \oplus r_0)$, then, we can simplify $\mathcal{E}(x_6)$ into $\widehat{\mathcal{E}(x_6)} = r_0 \odot r_0 \odot r_0$. Therefore, we can deduce that $\vdash \mathcal{E}(x_6) : \mathsf{SID}$ by applying the NoKey rule on $\widehat{\mathcal{E}(x_6)}$.

Algorithm 1: PMChecking(P, X_p, X_k, X_r, X_i)

```
Function PMChecking(P, X_p, X_k, X_r, X_i)
    foreach x ∈ X_i do
        if ⊢ E(x) : UKD is valid then
            if ⊢ Ê(x) : UKD is valid then
                if Ω(Ê(x)) exists then
                    Let ⊢ E(x) : τ be valid for valid ⊢ Ω(Ê(x)) : τ;
                else if ModelCountingBasedSolver(Ê(x))=SAT then
                    Let ⊢ E(x) : SDD be valid;
                else Let ⊢ E(x) : SID be valid;
            else Let ⊢ E(x) : τ be valid for valid ⊢ Ê(x) : τ;
```

4 Overall Algorithms

In this section, we present algorithms to check perfect masking and to compute the QMS values.

4.1 Perfect Masking Verification

Given a program P with the sets of public (X_p), secret (X_k), random (X_r) and internal (X_i) variables, PMChecking, given in Algorithm 1, checks whether P is perfectly masked or not. It iteratively traverses all the internal variables. For each variable $x \in X_i$, it first applies the type system to infer its distribution type. If $\vdash \mathcal{E}(x) : \tau$ for $\tau \neq \mathsf{UKD}$ is valid, then the result is conclusive. Otherwise, we will simplify the expression $\mathcal{E}(x)$ and apply the type inference to $\widehat{\mathcal{E}(x)}$.

If it fails to resolve the type of x and $\Omega(\widehat{\mathcal{E}(x)})$ does not exist, we apply the model-counting based (SMT-based or brute-force) approach outlined in Section 3.2, in particular, to check the expression $\widehat{\mathcal{E}(x)}$. There are two possible outcomes: either $\widehat{\mathcal{E}(x)}$ is SID or SDD. We enforce $\mathcal{E}(x)$ to have the same distributional type as $\widehat{\mathcal{E}(x)}$ which might facilitate the inference for other expressions.

Theorem 3. *P is perfectly masked iff $\vdash \mathcal{E}(x) : \mathsf{SDD}$ is not valid for any $x \in X_i$, when Algorithm 1 terminates.*

We remark that, if the model-counting is disabled in Algorithm 1 where UKD-typed variables are interpreted as potentially leaky, Algorithm 1 would degenerate to a type inference procedure that is fast and potentially more accurate than the one in [3], owing to the optimization introduced in Section 3.3.

4.2 QMS Computing

After applying Algorithm 1, each internal variable $x \in X_i$ is endowed by a distributional type of either SID (or RUD which implies SID) or SDD. In the former case, x is perfectly masked meaning observing x would gain nothing for

Algorithm 2: Procedure QMSComputing(P, X_p, X_k, X_r, X_i)

```
Function QMSComputing(P, X_p, X_k, X_r, X_i)
    PMChecking(P, X_p, X_k, X_r, X_i);
    foreach x ∈ X_i do
        if ⊢ ℰ(x) : SID is valid then QMS_x := 1;
        else
            if RVar(ℰ̂(x)) = ∅ then QMS_x := 0;
            else
                low := 0; high := 2^(n × |RVar(ℰ̂(x))|);
                while low < high do
                    mid := ⌈(low+high)/2⌉; q := mid / 2^(n × |RVar(ℰ̂(x))|);
                    if SMTSolver(Ψ̂_x^q) = SAT then high := mid − 1;
                    else low := mid;
                QMS_x := low / 2^(n × |RVar(ℰ̂(x))|);
```

side-channel attackers. In the latter case, however, x becomes a side-channel and it is natural to ask how many power traces are required to infer secret from x of which we have provided a measure formalized via QMS.

QMSComputing, given in Algorithm 2, computes $\mathtt{QMS}_x$ for each $x \in X_i$. It first invokes the function PMChecking for perfect masking verification. For SID-typed variable $x \in X_i$, we can directly infer that $\mathtt{QMS}_x$ is 1. For each leaky variable $x \in X_i$, we first check whether $\widehat{\mathcal{E}(x)}$ uses any random variables or not. If it does not use any random variables, we directly deduce that $\mathtt{QMS}_x$ is 0. Otherwise, we use either the brute-force enumeration or an SMT-based binary search to compute $\mathtt{QMS}_x$. The former one is trivial, hence not presented in Algorithm 2. The latter one is based on the fact that $\mathtt{QMS}_x = \frac{i}{2^{n \cdot |\mathsf{RVar}(\widehat{\mathcal{E}(x)})|}}$ for some integer $0 \leq i \leq 2^{n \cdot |\mathsf{RVar}(\widehat{\mathcal{E}(x)})|}$. Hence the while-loop in Algorithm 2 executes at most $\mathbf{O}(n \cdot |\mathsf{RVar}(\widehat{\mathcal{E}(x)})|)$ times for each x.

Our SMT-based binary search for computing QMS values is different from the one proposed by Eldib et al. [26,27]. Their algorithm considers Boolean programs *only* and computes QMS values by directly binary searching the QMS value q between 0 to 1 with a pre-defined step size ϵ ($\epsilon = 0.01$ in [26,27]). Hence, it only *approximate* the actual QMS value and the binary search iterates $\mathbf{O}(\log(\frac{1}{\epsilon}))$ times for each internal variable. Our approach works for more general arithmetic programs and computes the accurate QMS value.

5 Practical Evaluation

We have implemented our methods in a tool named QMVerif, which uses Z3 [23] as the underlying SMT solver (fixed size bit-vector theory). We conducted experiments perfect masking verification and QMS computing on both Boolean and arithmetic programs. Our experiments were conducted on a server with 64-bit Ubuntu 16.04.4 LTS, Intel Xeon CPU E5-2690 v4, and 256GB RAM.

Table 1. Results on masked Boolean programs for perfect masking verification.

Name	$\|X_i\|$	♯SDD	♯Count	QMVerif		SCInfer [50]
				SMT	B.F.	
P12	197k	0	0	2.9s	**2.7s**	3.8s
P13	197k	4.8k	4.8k	2m 8s	**2m 6s**	47m 8s
P14	197k	3.2k	3.2k	1m 58s	**1m 45s**	53m 40s
P15	198k	1.6k	3.2k	**2m 25s**	2m 43s	69m 6s
P16	197k	4.8k	4.8k	1m 50s	**1m 38s**	61m 15s
P17	205k	17.6k	12.8k	1m 24s	**1m 10s**	121m 28s

5.1 Experimental Results on Boolean Programs

We use the benchmarks from the publicly available cryptographic software implementations of [25], which consists of 17 Boolean programs (P1-P17). We conducted experiments on P12-P17, which are the regenerations of MAC-Keccak reference code submitted to the SHA-3 competition held by NIST. (We skipped tiny examples P1-P11 which can be verified in less than 1 second.) P12-P17 are transformed into programs in straight-line forms.

Perfect masking verification. Table 1 shows the results of perfect masking verification on P12-P17, where Columns 2-4 show basic statistics, in particular, they respectively give the number of internal variables, leaky internal variables, and internal variables which required model-counting based reasoning. Columns 5-6 respectively show the total time of our tool QMVerif using SMT-based and brute-force methods. Column 7 shows the total time of the tool SCInfer [50].

We can observe that: (1) our reduction heuristics significantly improve performance compared with SCInfer [50] (generally 22–104 times faster for imperfect masked programs; note that SCInfer is based on SMT model-counting), and (2) the performance of the SMT-based and brute-force counting methods for verifying perfect masking of Boolean programs is largely comparable.

Computing QMS. For comparison purposes, we implemented the algorithm of [25,24] for computing QMS values of leaky internal variables. Table 2 shows the results of computing QMS values on P13-P17 (P12 is excluded because it does not contain any leaky internal variable), where Column 2 shows the number of leaky internal variables, Columns 3-7 show the total number of iterations in the binary search (cf. Section 4.2), time, the minimal, maximal and average of QMS values using the algorithm from [25,24]. Similarly, Columns 8-13 shows statistics of our tool QMVerif, in particular, Column 9 (resp. Column 10) shows the time of using SMT-based (resp. brute-force) method. The time reported in Table 2 *excludes* the time used for perfect masking checking.

We can observe that (1) the brute-force method outperforms the SMT-based method for computing QMS values, and (2) our tool QMVerif using SMT-based methods takes significant less iterations and time, as our binary search step depends on the number of bits of random variables, but not a pre-defined value (e.g. 0.01) as used in [25,24]. In particular, the QMS values of leaky variables whose expressions contain no random variables, e.g., P13 and P17, do not need binary search.

Table 2. Results of masked Boolean programs for computing QMS Values.

Name	♯SDD	SC Sniffer [26,27]					QMVerif					
		♯Iter	Time	Min	Max	Avg.	♯Iter	SMT	B.F.	Min	Max	Arg.
P13	4.8k	480k	97m 23s	0.00	1.00	0.98	**0**	0	0	0.00	1.00	0.98
P14	3.2k	160k	40m 13s	0.51	1.00	0.99	9.6k	2m 56s	**39s**	0.50	1.00	0.99
P15	1.6k	80k	23m 26s	0.51	1.00	1.00	4.8k	1m 36s	**1m 32s**	0.50	1.00	1.00
P16	4.8k	320k	66m 27s	0.00	1.00	0.98	6.4k	1m 40s	**8s**	0.00	1.00	0.98
P17	17.6k	1440k	337m 46s	0.00	1.00	0.93	4.8k	51s	**1s**	0.00	1.00	0.94

Table 3. Results of masked arithmetic programs, where P.M.V. denotes perfect masking verification, B.F. denotes brute-force, 12 S.F. denotes that Z3 emits segmentation fault after verifying 12 internal variables.

Description	$\|X_i\|$	♯SDD	♯Count	P.M.V.		QMS		
				SMT	B.F.	SMT	B.F.	Value
SecMult [47]	11	0	0	≈0s	≈0s	-	-	1
Sbox (4) [22]	66	0	0	≈0s	≈0s	-	-	1
B2A [30]	8	**0**	**1**	17s	**2s**	-	-	1
A2B [30]	46	0	0	≈0s	≈0s	-	-	1
B2A [21]	82	0	0	≈0s	≈0s	-	-	1
A2B [21]	41	0	0	≈0s	≈0s	-	-	1
B2A [18]	11	**0**	**1**	**1m 35s**	10m 59s	-	-	1
B2A [10]	16	0	0	≈0s	≈0s	-	-	1
Sbox [47]	45	0	0	≈0s	≈0s	-	-	1
Sbox [48]	772	**2**	**1**	≈0s	≈0s	0.9s	≈0s	0
k^3	11	2	2	96m 59s	**0.2s**		32s	0.988
k^{12}	15	2	2	101m 34s	**0.3s**		27s	0.988
k^{15}	21	4	4	93m 27s (12 S.F.)	**28m 17s**		≈64h	0.988, 0.980
k^{240}	23	4	4	93m 27s (12 S.F.)	**30m 9s**		≈64h	0.988, 0.980
k^{252}	31	4	4	93m 27s (12 S.F.)	**32m 58s**		≈64h	0.988, 0.980
k^{254}	39	4	4	93m 27s (12 S.F.)	**30m 9s**		≈64h	0.988, 0.980

5.2 Experimental Results on Arithmetic Programs

We collect arithmetic programs which represent non-linear functions of masked cryptographic software implementations from the literature. In Table 3, Column 1 lists the name of the functions under consideration, where $k^3, \ldots, k^{254}$ are buggy fragments of first-order secure exponentiation [47] without the first RefreshMask function; A2B and B2A are shorthand for ArithmeticToBoolean and BooleanToArithmetic, respectively. Columns 2-4 show basic statistics. For all the experiments, we set $\mathbb{D} = \{0, \cdots, 2^8 - 1\}$.

Perfect masking verification. Columns 5-6 in Table 3 show the results of perfect masking verification on these programs using SMT-based and brute-force methods respectively.

We observe that (1) some UKD-typed variables (e.g., in B2A [30], B2A [18] and Sbox [48], meaning that the type inference is inconclusive in these cases) are resolved (as SID-type) by model-counting, and (2) on the programs (except B2A [18]) where model-counting based reasoning is required (i.e., ♯Count is non-zero), the brute-force method is significantly faster than the SMT-based method. In particular, for programs $k^{15}, \ldots, k^{254}$, Z3 crashed with segment fault after verifying 12 internal variables in 93min, while the brute-force method comfortably

returns the result. To further explain the performance of these two classes of methods, we manually examine these programs and find that the expressions of the UKD-typed variable (using type inference) in B2A [18] (where the SMT-based method is faster) only use exclusive-or ($\oplus$) operations and one subtraction ($-$) operation, while the expressions of the other UKD-typed variables (where the brute-force method is faster) involve the finite field multiplication ($\odot$).

We remark that the transformation oracle and meta-theorems (cf. Section 3.3) are only used for A2B [30] by manually utilizing the equations of Theorem 3 in [30]. We have verified the correctness of those equations by SMT solvers. In theory model-counting based reasoning could verify A2B [30]. However, in our experiments both SMT-based and brute-force methods failed to terminate in 3 days, though brute-force methods had verified more internal variables. For instance, on the expression $((2 \times r_1) \oplus (x - r) \oplus r_1) \wedge r$ where x is a private input and r, r_1 are random variables, Z3 cannot terminate in 2 days, while brute-force methods successfully verified in a few minutes. We also tested the SMT solver Boolector [39] (the winner of SMT-COMP 2018 on QF-BV, Main Track), which crashed with being out of memory. 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.

Computing QMS. Columns 7-9 in Table 3 show the results of computing QMS values, where Column 7 (resp. Column 8) shows the time of the SMT-based (resp. brute-force) method for computing QMS values (*excluding* the time for perfect masking verification) and Column 9 shows QMS values of all leaky variables (note that duplicated values are omitted).

6 Conclusion

We have proposed a hybrid approach combing type inference and model-counting to verify masked arithmetic programs against first-order side-channel attacks. The type inference allows an efficient, lightweight procedure to determine most observable variables whereas model-counting accounts for completeness, bringing the best of two worlds. We also provided model-counting based methods to quantify the amount of information leakage via side channels. We have presented the tool support QMVerif which has been evaluated on standard cryptographic benchmarks. The experimental results showed that our method significantly outperformed state-of-the-art techniques in terms of both accuracy and scalability.

Future work includes further improving SMT-based model counting techniques which currently provide no better, if not worse, performance than the naïve brutal-force approach. Furthermore, generalizing the work in the current paper to the verification of higher-order masking schemes remains to be a very challenging task.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.
2. 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, 2017.
3. 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)*, pages 457–485, 2015.
4. 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 *ACM Conference on Computer and Communications Security*, pages 116–129, 2016.
5. G. Barthe, S. Belaïd, P. Fouque, and B. Grégoire. maskverif: a formal tool for analyzing software and hardware masked implementations. *IACR Cryptology ePrint Archive*, 2018:562, 2018.
6. G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F. Standaert, and P. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology*, pages 535–566, 2017.
7. 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*, pages 1016–1027, 2014.
8. 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*, pages 140–158, 2014.
9. A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–310, 2013.
10. L. Bettale, J. Coron, and R. Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):22–45, 2018.
11. 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*, 2013:303, 2013.
12. E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 513–525, 1997.
13. 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)*, pages 71–93, 2018.
14. E. Bisi, F. Melzani, and V. Zaccaria. Symbolic analysis of higher-order side channel countermeasures. *IEEE Trans. Computers*, 66(6):1099–1105, 2017.

15. 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*, pages 321–353, 2018.
16. J. Breier, X. Hou, and Y. Liu. Fault attacks made easy: Differential fault analysis automation on assembly code. Cryptology ePrint Archive, Report 2017/829, 2017. https://eprint.iacr.org/2017/829.
17. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 398–412, 1999.
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)*, pages 93–114, 2017.
19. 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*, pages 65–82, 2018.
20. 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*, pages 69–81, 2012.
21. 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)*, pages 188–205, 2014.
22. J. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In *International Workshop on Fast Software Encryption*, pages 410–424, 2013.
23. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.
24. H. Eldib, C. Wang, and P. Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology*, 24(2):11, 2014.
25. H. Eldib, C. Wang, and P. Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 62–77, 2014.
26. H. Eldib, C. Wang, M. Taha, and P. Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*, pages 209:1–6, 2014.
27. 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*, 34(10):1558–1568, 2015.
28. H. Eldib, M. Wu, and C. Wang. Synthesis of fault-attack countermeasures for cryptographic circuits. In *International Conference on Computer Aided Verification*, pages 343–363, 2016.
29. 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*, 2017:711, 2017.
30. 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*, pages 3–15, 2001.

31. 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.
32. Y. Ishai, A. Sahai, and D. A. Wagner. Private circuits: Securing hardware against probing attacks. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 463–481, 2003.
33. P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 104–113, 1996.
34. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 388–397, 1999.
35. P. Malacaria and J. Heusser. Information theory and security: Quantitative information flow. In *Formal Methods for Quantitative Aspects of Programming Languages, 10th International School on Formal Methods for the Design of Computer, Communication and Software Systems, (SFM)*, pages 87–134, 2010.
36. 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)*, pages 313–327, 2018.
37. A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 111–124, 2011.
38. 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)*, pages 58–75, 2012.
39. A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).
40. 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.
41. 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)*, pages 387–400, 2016.
42. 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)*, pages 328–342, 2017.
43. 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)*, pages 283–292, 2014.
44. 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)*, pages 105–108, 2014.
45. 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*, pages 142–159, 2013.

46. J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *Proceedings of International Conference on Research in Smart Cards (E-smart)*, pages 200–210, 2001.
47. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427, 2010.
48. K. Schramm and C. Paar. Higher order masking of the AES. In *Proceedings of the RSA Conference on Topics in Cryptology (CT-RSA)*, pages 208–225, 2006.
49. 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)*, pages 31–46, 2016.
50. 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 (CAV), Held as Part of the Federated Logic Conference*, pages 157–177, 2018.