Verifying and Quantifying Side-channel Resistance of Masked Software Implementations

PENGFEI GAO, JUN ZHANG, and FU SONG, ShanghaiTech University, China
CHAO WANG, University of Southern California, USA

Power side-channel attacks, capable of deducing secret data using statistical analysis, have become a serious threat. Random masking is a widely used countermeasure for removing the statistical dependence between secret data and side-channel information. Although there are techniques for verifying whether a piece of software code is perfectly masked, they are limited in accuracy and scalability. To bridge this gap, we propose a refinement-based method for verifying masking countermeasures. Our method is more accurate than prior type-inference-based approaches and more scalable than prior model-counting-based approaches using SAT or SMT solvers. Indeed, our method can be viewed as a gradual refinement of a set of type-inference rules for reasoning about distribution types. These rules are kept *abstract* initially to allow fast deduction and then made *concrete* when the abstract version is not able to resolve the verification problem. We also propose algorithms for quantifying the amount of side-channel information leakage from a software implementation using the notion of quantitative masking strength. We have implemented our method in a software tool and evaluated it on cryptographic benchmarks including AES and MAC-Keccak. The experimental results show that our method significantly outperforms state-of-the-art techniques in terms of accuracy and scalability.

CCS Concepts: • **Security and privacy** → **Side-channel analysis and countermeasures**; • **Software and its engineering** → **Software verification**; **Automated static analysis**;

Additional Key Words and Phrases: Differential power analysis, perfect masking, type inference, quantitative masking strength, satisfiability modulo theory (SMT), cryptographic software, AES, MAC-Keccak

ACM Reference format:
Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and Quantifying Side-channel Resistance of Masked Software Implementations. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 16 (July 2019), 32 pages.
https://doi.org/10.1145/3330392

This article extends the results presented in Reference [86]. In addition to more detailed descriptions of the algorithm and related work and additional data in the experimental results, this article contains new materials on verifying *quantitative masking strength* (Section 5, Section 6.3 and Section 6.4).

P. Gao is also with University of Chinese Academy of Sciences.
P. Gao is also with Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences.
J. Zhang is also with University of Chinese Academy of Sciences.
J. Zhang is also with Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences.

This work was supported primarily by the National Natural Science Foundation of China (NSFC) Grants No. 61532019 and No. 61761136011. Chao Wang was supported by the U.S. National Science Foundation (NSF) Grant No. CNS-1617203.

Authors' addresses: P. Gao, J. Zhang, and F. Song (corresponding author), ShanghaiTech University, 393 Middle Huaxia Rd, Shanghai 201210, China; emails: {gaopf, zhangjun, songfu}@shanghaitech.edu.cn; C. Wang, University of Southern California, 941 Bloom Walk Rd, Los Angeles, CA, 90089; email: wang626@usc.edu.

1049-331X/2019/07-ART16 $15.00
https://doi.org/10.1145/3330392

1 INTRODUCTION

Cryptographic algorithms are widely used in embedded computing devices, including SmartCards, to form the backbone of their security mechanisms. In general, security is established by assuming that the adversary may have access to the input, output, and internal structure of the implementation but not the secret data (e.g., cryptographic keys and random numbers). Unfortunately, in practice, attackers may recover the secret data by analyzing physical information leaked through side channels. These so-called *side-channel attacks* exploit the statistical dependence between secret data and non-functional properties of a computing device such as the execution time [54], power consumption [55], and electromagnetic radiation [73]. Among them, *differential power analysis* is a popular and effective class of attacks [44, 60]. For example, the power consumption of a device executing the instruction $c = p \wedge k$, where $\wedge$ denotes the logical AND, depends on the value of the secret k, which can be reliably deduced using differential power analysis.

Numerous countermeasures have been developed to thwart side-channel attacks; for power analysis-based attacks, in particular, *masking* remains the most widely used technique. Masking is a randomization technique aimed to remove the statistical dependence between secret data and the power consumption. In principle, given a masking order d, masking makes use of a secret-sharing scheme to split the secret data into $d + 1$ shares such that any subset of at most d shares is *statistically independent* on the secret data. Here, d is a security parameter: order-d masking is secure against order-d attacks, where the adversary may have access to at most d shares, but the knowledge of all $(d + 1)$ shares still allows for recovery of the secret data. For example, the masked value of a secret bit, k, can be computed using an exclusive-or ($\oplus$) operation and a random bit, r, in the form of $k \oplus r$; correspondingly, the bit can be recovered by demasking: $(k \oplus r) \oplus r = k \oplus (r \oplus r) = k$.

Many masked implementations have been proposed over the years, e.g., for AES or its non-linear S-boxes [22, 53, 75, 76]. In general, masking for linear functions, where *linear* is defined in terms of the exclusive-or ($\oplus$) operation, is straightforward [40]. However, masking for non-linear functions, which are actually used in almost all cryptographic algorithms, is difficult, because the process is both labor intensive and error prone. Indeed, there have been published implementations [75, 76] later shown to be incorrect [34, 35]. Therefore, formally verifying and quantifying the side-channel resistance of these masking countermeasures are important.

Previously, there are two types of formal verification methods for masking countermeasures [82]: One is type-inference based [15, 62] and the other is model-counting based [39, 40]. Type-inference-based methods [15, 62] are fast and sound, meaning that they can quickly prove that the computation is leakage free, e.g., when the result is syntactically independent of the secret data or has been masked by *fresh* random variables—random variables that are not used elsewhere in cryptographic software. However, syntactic type inference is *not* complete in that it may report *false positives*. In contrast, model-counting-based methods [39, 40] are both sound and complete: They can decide, with certainty, whether the computation is statistically independent of the secret data [22]. However, due to the inherent complexity of the model-counting approach, they can be slow in practice.

The aforementioned gap, in terms of accuracy and scalability, has not yet been bridged by more recent approaches [8, 9, 20, 33, 65]. For example, Barthe et al. [8, 9] proposed some inference rules to prove masking countermeasures based on the observation that certain operators (e.g., XOR) are *invertible*: Purely algebraic laws can be used to normalize expressions of computation results to apply the rules of invertible functions. However, since this normalization is costly, it is applied to each expression only once. Coron [33] proposed two alternative approaches to improve efficiency, using elementary circuit transformations instead of expression normalization. Ouahma

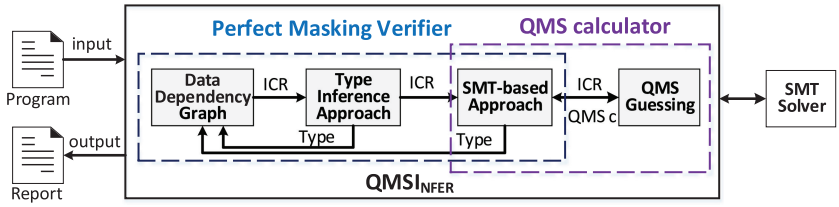

Fig. 1. Overview of our refinement-based approach QMSInfer, where "ICR" denotes the intermediate computation result.

et al. [65] introduced a similar, linear-time algorithm based on finer-grained syntactic inference rules. Another similar idea was explored by Bisi et al. [20] for analyzing higher-order masking; like the methods in References [8, 33, 65], however, the method is not complete and does not consider non-linear operators that are common in cryptographic software.

Furthermore, all the existing approaches mentioned above focus on verifying whether implementations are perfectly masked. Although perfect masking is ideal, it is not always achievable in practice, e.g., when only a limited number of random variables are allowed for efficiency considerations [63]. In such cases, there will be negative impact on side-channel resistance, and, naturally, one wants to quantify how severe the impact is. For instance, one possible measure is the resource the attacker needs to invest to infer the secret data from the side channel. For this purpose, we adopt the notion of *Quantitative Masking Strength (QMS)*, with which a correlation of the number of power measurement traces needed for attackers to deduce secret data has been established empirically [41, 42].

Our contribution. We propose a refinement-based approach, named QMSInfer, to bridge the gap between prior techniques that are either fast but inaccurate or accurate but slow. Figure 1 depicts the overall flow, where the input of QMSInfer consists of the masked program and its input variables marked as *public*, *private*, or *random*, and the output of QMSInfer is a security report. Inside QMSInfer, there are two main components: *perfect masking verifier* and *QMS calculator*. To verify perfect masking, we first transform the program to an intermediate representation: the data dependency graph (DDG). Then, we traverse the DDG in a topological order to infer a *distribution type* for each intermediate computation result to prove that it is leakage free.

If perfect masking cannot be proved this way using the distribution type, then we invoke an SMT solver-based refinement procedure, which leverages either satisfiability-checking (SAT) or model-counting (SAT#) to prove the leakage-free property. The model-counting-based method is complete in that it can decide, with certainty, whether the result is perfectly masked. Regardless of whether it is perfectly masked, the result is fed back to improve the type-inference system. Finally, based on the refined type-inference result, we continue to analyze the side-channel resistance property of other intermediate computation results. If any of the intermediate computation results is not perfectly masked, then we compute its QMS value [41, 42] via the SMT solver-based approach to quantify the amount of information leakage through the power side channel.

Thus, QMSInfer can be viewed as a synergistic integration of a rule-based approach for inferring distribution types and an SMT-based approach for refining these types. Our type-inference rules (Section 3) are inspired by Barthe et al. [8] and Ouahma et al. [65], who also infer distribution types, but there is a crucial difference: Their inference rules are syntactic with fixed accuracy, i.e., relying solely on syntactic information of the program, whereas ours are *semantic* and the accuracy can be gradually improved with the aid of our SMT solver-based analysis. At a high level, our gradually refined semantic type-inference rules subsume their syntactic type-inference rules.

The main advantage of using type inference is the ability to *quickly* obtain sound proofs: When there is no leak in the computation, often the type system can produce a proof quickly; furthermore, the result is always conclusive. However, if type inference fails to produce a proof, the verification problem remains unresolved. Thus, to be complete, we leverage the SMT-based model-counting approach to resolve these *left-over* verification problems. Here, solvers are used to check either the satisfiability (SAT) of a logical formula or counting its satisfying solutions (SAT#), the later of which, although expensive, is powerful enough to completely decide whether the computation is leakage free. Finally, by feeding solver results back to the type system, we can gradually improve its accuracy. Thus, overall, our method is both sound and complete.

Our QMS calculator is inspired by Eldib et al. [41, 42], who showed empirically that the number of measurement traces required by a differential power analysis– (DPA) based attack to successfully deduce the secret key is reflected by the QMS value. However, there are two crucial differences between our work and that of Eldib et al. [41, 42]: First, our approach computes the accurate QMS value of each intermediate computation result, while their approach only computes an approximation of the QMS value; second, our approach is tightly integrated with our perfect masking verifier, which allows us to skip the computation of QMS values for all the perfectly masked intermediate computation results, while their approach may compute, unnecessarily, the QMS values of perfectly masked intermediate computation results.

We have evaluated QMSInfer on a set of publicly available benchmarks [39, 40], which implement various cryptographic algorithms such as AES and MAC-Keccak. Our experiments show that QMSInfer is both effective in obtaining proofs quickly and scalable for handling realistic applications. Specifically, it can resolve most of the verification subproblems using type inference and, as a result, satisfiability-checking- (SAT) based analysis is needed only for a few left-over cases. Only in rare cases, the most heavyweight, model-counting-based analysis (SAT#) needs to be invoked.

To sum up, the main contributions of this work are as follows:

- We propose a new semantic type-inference approach for verifying masking countermeasures. It is sound and efficient for obtaining proofs.
- We propose a novel method for refining the type-inference system using an SMT solver-based analysis to ensure that the overall method is both sound and complete.
- We propose a new algorithm to compute, for intermediate results that are not perfectly masked, their quantitative masking strength (QMS) values.
- We implement the proposed techniques in a tool named QMSInfer and demonstrate its effectiveness on cryptographic software benchmarks.

The source code of QMSInfer and the benchmarks used in this work have been made available at http://faculty.sist.shanghaitech.edu.cn/faculty/songfu/Projects/SCInfer/qmsInfer-master.zip.

The remainder of this article is organized as follows. After reviewing the basics in Section 2, we present our semantic type-inference system in Section 3. We present our refinement-based method for verifying perfect masking and computing QMS values in Section 4 and Section 5, respectively. We present our experimental results in Section 6 and comparison with the related work in Section 7. Finally, we give our conclusions in Section 8.

2 PRELIMINARIES

In this section, we define the type of programs considered in this work and then review the basics of side-channel attacks and masking countermeasures.

2.1 Probabilistic Boolean Programs

Following the notation used in References [22, 39, 40], we assume that the program P for implementing a cryptographic function is in the form of

$$X_c \leftarrow P(X_p, X_k),$$

where X_p is the plaintext, X_k is the secret key, and X_c is the ciphertext. Inside P, random variable X_r may be used to mask the secret key while maintaining the input-output behavior of P. Therefore, P can be regarded as a probabilistic program. Since loops, function calls, and branches in cryptographic implementations can be removed via automated program rewriting [39, 40] and integer variables can be represented by bit-vectors, for verification purposes, we assume that P is a straight-line probabilistic Boolean program, where each instruction has a unique label and at most two operands.

Let $X = X_k \uplus X_r \uplus X_p \uplus X_c$ be the set of Boolean variables used in P, where X_k are the secret bits, X_r are the random bits, X_p are the public bits, and X_c are the variables storing intermediate results. In addition, the program uses a set of operators including negation ($\neg$), and ($\wedge$), or ($\vee$), and exclusive-or ($\oplus$). A *computation* of P is a sequence of intermediate computation results: $c_1 \leftarrow \mathbf{I}_1(X_p, X_k, X_r); \ldots; c_n \leftarrow \mathbf{I}_n(X_p, X_k, X_r)$, where, for each $1 \leq i \leq n$, the computation of $\mathbf{I}_i$ is expressed in terms of X_p, X_k, and X_r. Each random bit in X_r is uniformly distributed in $\{0, 1\}$; the sole purpose of using them in P is to ensure that $c_1, \ldots c_n$ are statistically independent of the secret X_k.

Data dependency graph (DDG). Our internal representation of a program P is a graph $\mathcal{G}_P = (N, E, \lambda)$, where N is the set of nodes, E is the set of edges, and λ is a labeling function.

- $N = L \uplus L_X$, where L is the set of instruction labels and L_X is the set of terminal nodes: $l_x \in L_X$ corresponds to a variable or constant $x \in X_k \cup X_r \cup X_p \cup \{0, 1\}$.
- $E \subseteq N \times N$ contains edge $(l, l') \in L \times L$ if and only if $l : c = x \circ y$, where either x or y is defined by l' (i.e., use-define relation) or $l : c = \neg x$, where x is defined by l', and contains edge $(l, l_x) \in L \times L_x$ if and only if $l : c = e$ and the input variable or constant x is used in e;
- λ maps each $l \in N$ to a pair (val, op): $\lambda(l) = (c, \circ)$ for $l : c = x \circ y$; $\lambda(l) = (c, \neg)$ for $l : c = \neg x$; and $\lambda(l_x) = (x, \perp)$ for each terminal node l_x.

We may use $\lambda_1(l) = c$ and $\lambda_2(l) = \circ$ to denote the first and second elements of the pair $\lambda(l) = (c, \circ)$, respectively. We may also use $l.\texttt{lft}$ to denote the left child of l, and $l.\texttt{rgt}$ to denote the right child if it exists. A subtree rooted at node l corresponds to an intermediate computation result. When the context is clear, we may use the following terms exchangeably: a node l, the subtree T rooted at l, and the intermediate computation result c such that c is $\lambda_1(l)$. Let $|P|$ denote the total number of nodes in the DDG $\mathcal{G}_P$.

Example 2.1. Figure 2 shows an example where $X_k = \{k\}$, $X_r = \{r_1, r_2, r_3\}$, $X_c = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, and $X_p = \emptyset$. The left-hand part is the original program written in a C-like language, except that $\oplus$ denotes XOR and $\wedge$ denotes AND. The right-hand part is the corresponding DDG. It is easy to see that:

$$\begin{aligned}
c_3 &= c_2 \oplus c_1 = k \oplus r_1 \\
c_4 &= c_3 \oplus c_2 = k \oplus r_2 \\
c_5 &= c_4 \oplus r_1 = k \oplus r_1 \oplus r_2 \\
c_6 &= c_5 \wedge r_3 = (k \oplus r_1 \oplus r_2) \wedge r_3
\end{aligned}$$

```
1  bool compute(bool r1, bool r2,
2               bool r3, bool k)
3  {
4    bool c1, c2, c3, c4, c5, c6;
5    c1 = k ⊕ r2;
6    c2 = r1 ⊕ r2;
7    c3 = c2 ⊕ c1;
8    c4 = c3 ⊕ c2;
9    c5 = c4 ⊕ r1;
10   c6 = c5 ∧ r3;
11   return c6;
12 }
```

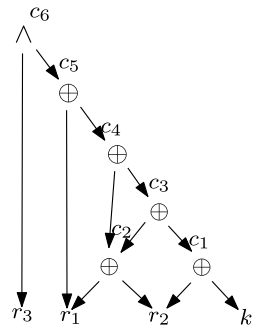

Fig. 2. An example for masking countermeasure: left-hand part is a C-like program and right-hand part is its DDG, where r_1, r_2, and r_2 are random variables, k is a secret variable, $\oplus$ denotes logical XOR, and $\wedge$ denotes logical AND.

Let $\mathsf{Supp} : N \to X_k \cup X_r \cup X_p$ be a *supporting variable function* mapping each node l to its support variables. That is, $\mathsf{Supp}(l) = \emptyset$ if $\lambda_1(l) \in \{0, 1\}$; $\mathsf{Supp}(l) = \{x\}$ if $\lambda_1(l) = x \in X_k \cup X_r \cup X_p$; and $\mathsf{Supp}(l) = \mathsf{Supp}(l.\mathtt{lft}) \cup \mathsf{Supp}(l.\mathtt{rgt})$ otherwise. The function returns a set of variables that $\lambda_1(l)$ depends upon syntactically. We define the *supporting random variable function* $\mathsf{SuppR} : N \to X_r$ such that $\mathsf{SuppR}(l) = \mathsf{Supp}(l) \cap X_r$ for each $l \in N$.

Given an intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$, we say that it is *semantically dependent on* a variable $r \in X$ if and only if there exist two assignments, π_1 and π_2, such that $\pi_1(r) \neq \pi_2(r)$ and $\pi_1(x) = \pi_2(x)$ for every $x \in X \setminus \{r\}$, and the values of $\mathbf{I}(X_p, X_k, X_r)$ differ under π_1 and π_2.

Let $\mathsf{SemdR} : N \to X_r$ be a *semantically dependent random variable function* such that $\mathsf{SemdR}(l)$ denotes the set of *random variables* upon which the intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ of l semantically depends. Thus, $\mathsf{SemdR}(l) \subseteq \mathsf{SuppR}(l)$, and for each $r \in \mathsf{SuppR}(l) \setminus \mathsf{SemdR}(l)$, we know $\lambda_1(l)$ is semantically independent of r. More importantly, there is often a gap between $\mathsf{SuppR}(l)$ and $\mathsf{SemdR}(l)$, namely $\mathsf{SemdR}(l) \subset \mathsf{SuppR}(l)$, which is why our gradual refinement of semantic type inference rules can outperform methods based solely on syntactic type inference.

Example 2.2. Consider the node c_4 in Figure 2: We have $\mathsf{Supp}(c_4) = \{r_1, r_2, k\}$, $\mathsf{SemdR}(c_4) = \{r_2\}$, and $\mathsf{SuppR}(c_4) = \{r_1, r_2\}$. Furthermore, if the random bits are uniformly distributed in $\{0, 1\}$, then c_4 is both *uniformly distributed* and *statistically secret independent* (cf. Section 2.2).

2.2 Side-channel Attacks and Perfect Masking

We assume that the adversary has access to the public input X_p and output X_c, but not the secret X_k and random variable X_r, of the program $X_c \leftarrow P(X_p, X_k)$. However, the adversary may have access to side-channel leaks that reveal the joint distribution of at most d intermediate computation results $c_1, \ldots c_d$ (e.g., via differential power analysis [55]). Under these assumptions, the goal of the adversary is to deduce information of X_k. To model the leakage of each instruction, we consider a widely used, value-based model, called the Hamming Weight (HW) model; other power leakage models such as the Hamming Distance model [6] can be used similarly [8].

Let $[n]$ denote the set $\{1, \ldots, n\}$ of natural numbers where $n \geq 1$. We call a set with d elements a *d-set*. Given concrete values (V_p, V_k) for (X_p, X_k) and a d-set $\{c_1, \ldots, c_d\}$ of intermediate computation results, we use $\mathrm{D}_{V_p, V_k}(c_1, \ldots c_d)$ to denote the joint distribution of $c_1, \ldots, c_d$ induced by instantiating X_p and X_k with concrete values V_p and V_k, respectively. We use $\mathrm{D}_{V_p, V_k}(c_1, \ldots c_d)(v_1, \ldots, v_d)$ to denote the probability of the intermediate computation results $c_1, \ldots c_d$, respectively, being evaluated to $v_1, \ldots, v_d$. Formally, for each vector of values $v_1, \ldots, v_d$

in the probability space $\{0,1\}^d$, we have $\mathrm{D}_{V_p,V_k}(c_1,\ldots c_d)(v_1,\ldots,v_d) =$

$$\frac{\left|\left\{V_r \in \{0,1\}^{|X_r|} : \begin{array}{c} v_1 = \mathbf{I}_1(X_p = V_p, X_k = V_k, X_r = V_r), \\ \ldots, \\ v_d = \mathbf{I}_d(X_p = V_p, X_k = V_k, X_r = V_r) \end{array}\right\}\right|}{2^{|X_r|}},$$

where for every $1 \le j \le d$, the predicate $v_j = \mathbf{I}_j(X_p = V_p, X_k = V_k, X_r = V_r)$ holds if and only if the intermediate computation result $\mathbf{I}_j(X_p, X_k, X_r)$ is evaluated to v_j when instantiating X_p, X_k, and X_r with concrete values V_p, V_k, and V_r, respectively.

Definition 2.3. We say a d-set $\{c_1,\ldots,c_d\}$ of intermediate computation results is

- *uniformly distributed* if $\mathrm{D}_{V_p,V_k}(c_1,\ldots,c_d)$ is a uniform distribution for any concrete values V_p and V_k.
- *(statistically) secret independent* if $\mathrm{D}_{V_p,V_k}(c_1,\ldots,c_d) = \mathrm{D}_{V_p,V'_k}(c_1,\ldots,c_d)$ for any pairs of concrete values (V_p, V_k) and (V_p, V'_k).

Note that there is a difference between them: An uniformly distributed d-set is always secret independent but a secret independent d-set is not always uniformly distributed.

Definition 2.4. A program P is *order-d perfectly masked* if every d'-set of intermediate computation results of P such that $d' \le d$ is secret independent. When P is first-order perfectly masked, we may simply say it is perfectly masked.

To decide whether P is order-d perfectly masked, the assumption of masking is invalidated if there exist a d'-set and two pairs (V_p, V_k) and (V_p, V'_k) such that $\mathrm{D}_{V_p,V_k}(c_1,\ldots,c_{d'}) \neq \mathrm{D}_{V_p,V'_k}(c_1,\ldots,c_{d'})$ for some $d' \le d$. In this context, the main challenge is computing $\mathrm{D}_{V_p,V_k}(c_1,\ldots,c_{d'})$, which is essentially a *model-counting* (SAT#) problem. In the remainder of this article, we mainly focus on (first-order) perfect masking.

Gap in current state of knowledge. Existing methods for verifying masking countermeasures are either *fast but inaccurate*, e.g., when they rely solely on syntactic type inference (syntactic information provided by `SuppR` in Section 2.1) or *accurate but slow*, e.g., when they rely solely on model-counting. In contrast, our method gradually refines a set of semantic type inference rules (i.e., using `SemdR` instead of `SuppR` as defined in Section 2.1), where constraint solvers (SAT and SAT#) are used on demand to resolve ambiguity and improve the accuracy of type inference. As a result, it can achieve the best of both worlds.

2.3 Quantitative Masking Strength

When a program is leaky, it is important to quantify the amount of information leakage from the software through the side channel. In this work, we adopt a notion proposed by Eldib et al. [41, 42], named *Quantitative Masking Strength* (QMS), to quantify the strength of a masking countermeasure against first-order, DPA-based attacks.

Definition 2.5. The quantitative masking strength (QMS) of an intermediate computation result $\mathbf{I}(X_p, X_k, X_r)$ in a program P, denoted $\mathtt{QMS}_{\mathbf{I}}$, is defined as follows:

$$1 - \max_{V_p,V_k,V'_k}\Big(E(\mathbf{I}(V_p/X_p, V_k/X_k)) - E(\mathbf{I}(V_p/X_p, V'_k/X_k))\Big),$$

where $E(\circ)$ is the expected value of random event $\circ$. Intuitively, the larger $\mathtt{QMS}_{\mathbf{I}}$ is, the less information is leaked.

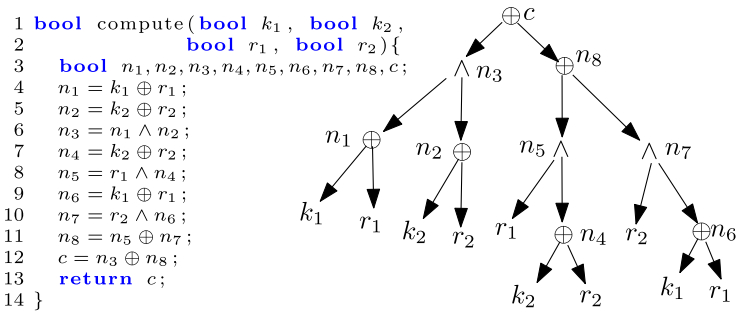

Fig. 3. An example for illustrating QMS: left-hand part is the C-like program and right-hand part is its data dependency graph, where r_1 and r_2 are random variables, k_1 and k_2 are secret variables.

It is easy to observe that the notion of QMS subsumes the notion of (first-order) perfect masking, namely, an intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ is perfectly masked iff $\mathtt{QMS}_{\mathbf{I}} = 1$.

For intermediate computation results that are not perfectly masked, QMS can be used to quantify the amount of information leakage through side channels. It is also an estimation of the degree of security of the program against side-channel attacks. The correlation between the QMS value and the number of measurement traces required by DPA-based attacks has been validated empirically by Eldib et al. [41, 42]. Specifically, they showed that the number of traces needed to deduce the secret key is exponentially dependent on the QMS value. Therefore, we use QMS as a formal quantitative measure.

In this work, when given a program P with some leaky nodes, we want to compute the actual QMS values of these nodes.

Example 2.6. Consider the program shown in Figure 3, which is taken from Reference [40]. This program is a masked version of $c \leftarrow k_1 \wedge k_2$ using the masking scheme of Blömer et al. [22], where k_1 and k_2 are the secrets, r_1 and r_2 are random variables that are used to make the power consumption of the computation of c statistically independent of the values of k_1 and k_2. The result c is logically equivalent to $(k_1 \wedge k_2) \oplus (r_1 \wedge r_2)$. The desired value $k_1 \wedge k_2$ could be obtained by applying the demasking function $c \oplus (r_1 \wedge r_2)$ (not shown in Figure 3), as $((k_1 \wedge k_2) \oplus (r_1 \wedge r_2)) \oplus (r_1 \wedge r_2) \equiv k_1 \wedge k_2$.

It is easy to see that n_i is first-order perfectly masked for all $i \in \{1, \ldots, 7\}$ (implying that $\mathtt{QMS}_{n_i} = 1$). The probability for n_8 to be logical one is 0 if $k_1k_2 = 00$ and is $\frac{1}{2}$ otherwise. Therefore, n_8 is a leaky node, and $\mathtt{QMS}_{n_8} = \frac{1}{2}$. Similarly, we can deduce that c is a leaky node and $\mathtt{QMS}_c = \frac{1}{2}$.

3 THE SEMANTIC TYPE INFERENCE SYSTEM

In this section, we present our *semantic* type-inference system. We first introduce our distribution types together with some auxiliary data structures; then, we present our inference rules.

3.1 The Type System

Let $\mathsf{T} = \{\mathtt{CST}, \mathtt{RUD}, \mathtt{SID}, \mathtt{NPM}, \mathtt{UKD}\}$ be the set of (distribution) types for intermediate computation results, where $[\![c]\!]$ denotes the type of $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$. Specifically,

- $[\![c]\!] = \mathtt{CST}$ means c is a constant, which implies that it is side-channel leak-free;
- $[\![c]\!] = \mathtt{RUD}$ means c is randomized to uniform distribution, and hence leak-free;
- $[\![c]\!] = \mathtt{SID}$ means c is statistically secret independent, i.e., leak-free;
- $[\![c]\!] = \mathtt{NPM}$ means c is not perfectly masked and thus has leaks; and
- $[\![c]\!] = \mathtt{UKD}$ means c has an unknown distribution.

Definition 3.1. Let $\mathsf{unq} : N \to X_r$ and $\mathsf{dom} : N \to X_r$, respectively, be *unique supporting random variable function* and *dominant random variable function* such that (i) for each terminal node $l \in L_X$ if $\lambda_1(l) \in X_r$, then $\mathsf{unq}(l) = \mathsf{dom}(l) = \lambda_1(l)$; otherwise, $\mathsf{unq}(l) = \mathsf{dom}(l) = \emptyset$; and (ii) for each internal node $l \in L$, we have

- $\mathsf{unq}(l) = (\mathsf{unq}(l.\mathtt{lft}) \cup \mathsf{unq}(l.\mathtt{rgt})) \setminus (\mathsf{SuppR}(l.\mathtt{lft}) \cap \mathsf{SuppR}(l.\mathtt{rgt}))$;
- $\mathsf{dom}(l) = (\mathsf{dom}(l.\mathtt{lft}) \cup \mathsf{dom}(l.\mathtt{rgt})) \cap \mathsf{unq}(l)$ if $\lambda_2(l) = \oplus$; but $\mathsf{dom}(l) = \emptyset$ otherwise.

Intuitively, the function unq that returns, for each $l \in N$, the set of random variables to which the DDG has a unique path from l and the function dom returns the set of random variables, each of which guarantees $\lambda_1(l)$ has been perfectly masked. Both $\mathsf{unq}(l)$ and $\mathsf{dom}(l)$ are computable in time that is linear in $|P|$ [65]. Following the proofs in References [8, 65], it is easy to reach this observation: Given an intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ that corresponds to a subtree rooted at l, the following statements hold:

(1) if $|\mathsf{dom}(l)| \neq \emptyset$, then $[\![c]\!] = \mathtt{RUD}$;
(2) if $[\![c]\!] = \mathtt{RUD}$, then $[\![\neg c]\!] = \mathtt{RUD}$;
(3) if $[\![c]\!] = \mathtt{SID}$, then $[\![\neg c]\!] = \mathtt{SID}$;
(4) if $r \notin \mathsf{SemdR}(l)$ for a bit $r \in X_r$, then $[\![r \oplus c]\!] = \mathtt{RUD}$;
(5) for every $c' \leftarrow \mathbf{I}'(X_p, X_k, X_r)$ that corresponds to a subtree rooted at l', if $\mathsf{SemdR}(l) \cap \mathsf{SemdR}(l') = \emptyset$ and $[\![c]\!] = [\![c']\!] = \mathtt{SID}$, then $[\![c \circ c']\!] = \mathtt{SID}$.

When the context is clear, we may use $[\![l]\!]$ and $[\![c]\!]$ exchangeably for an intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ that corresponds to a subtree rooted at the node l.

Figure 4 shows our type inference rules that concretize this observation. Each rule is given in the form of

$$\textsc{RuleName}\frac{\mathit{Hypothesis}_1 \quad \cdots \quad \mathit{Hypothesis}_k}{\mathit{Conclusion}}$$

where RuleName denotes the name of the rule, $\mathrm{Hypothesis}_1 \cdots \mathrm{Hypothesis}_k$ are hypothesises of the rule, and Conclusion, which holds if all the hypothesises hold.

When multiple rules could be applied to a node $l \in N$, we always choose the rules that can lead to $[\![l]\!] = \mathtt{RUD}$. If no rule is applicable at l, then we set $[\![l]\!] = \mathtt{UKD}$. The correctness of these inference rules is obvious by definition.

Remark that our type proof system currently will not annotate NPM to nodes. We will resolve UKD into either SID or NPM in Section 4 using the SMT-based analyses.

Theorem 3.2. *For every intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$,*

- *if $[\![c]\!] = \mathtt{RUD}$, then c is uniformly distributed, and hence perfectly masked;*
- *if $[\![c]\!] = \mathtt{SID}$, then c is guaranteed to be leakage-free.*

To improve efficiency, our inference rules may be applied twice, first using the SuppR function, which extracts syntactic information from the program (cf. Section 2.1) and then using the SemdR function, which is slower to compute but also significantly more accurate. Since $\mathsf{SemdR}(l) \subseteq \mathsf{SuppR}(l)$ for all $l \in N$, this is always sound. Moreover, the type inference is invoked for the second time only if, after the first time, $[\![l]\!]$ remains UKD.

Example 3.3. When using type inference with SuppR on the example in Figure 2, we have

$$[\![r_1]\!] = [\![r_2]\!] = [\![r_3]\!] = [\![c_1]\!] = [\![c_2]\!] = [\![c_3]\!] = \mathtt{RUD},\ [\![k]\!] = [\![c_4]\!] = [\![c_5]\!] = [\![c_6]\!] = \mathtt{UKD}.$$

When using type inference with SemdR (for the second time), we have

$$[\![r_1]\!] = [\![r_2]\!] = [\![r_3]\!] = [\![c_1]\!] = [\![c_2]\!] = [\![c_3]\!] = [\![c_4]\!] = [\![c_5]\!] = \mathtt{RUD},\ [\![k]\!] = \mathtt{UKD},\ [\![c_6]\!] = \mathtt{SID}.$$

$$\textsc{Leaf}_1\ \frac{\lambda_1(l) \in X_r}{[\![l]\!] = \mathsf{RUD}} \qquad \textsc{Leaf}_2\ \frac{\lambda_1(l) \in X_p \cup X_k}{[\![l]\!] = \mathsf{UKD}} \qquad \textsc{Leaf}_3\ \frac{\lambda_1(l) \in \{0,1\}}{[\![l]\!] = \mathsf{CST}}$$

$$\textsc{Xor-Rud}_1\ \frac{\lambda_2(l) = \oplus \quad [\![l.\mathtt{lft}]\!] = \mathsf{RUD} \quad \mathsf{dom}(l.\mathtt{lft}) \setminus \mathsf{SemdR}(l.\mathtt{rgt}) \neq \emptyset}{[\![l]\!] = \mathsf{RUD}}$$

$$\textsc{Xor-Rud}_2\ \frac{\lambda_2(l) = \oplus \quad [\![l.\mathtt{rgt}]\!] = \mathsf{RUD} \quad \mathsf{dom}(l.\mathtt{rgt}) \setminus \mathsf{SemdR}(l.\mathtt{lft}) \neq \emptyset}{[\![l]\!] = \mathsf{RUD}}$$

$$\textsc{AO-Sid}_1\ \frac{\begin{array}{c}\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{rgt}]\!] \notin \{\mathsf{UKD}, \mathsf{NPM}\} \quad [\![l.\mathtt{lft}]\!] = \mathsf{RUD} \\ \mathsf{SemdR}(l.\mathtt{lft}) \cap \mathsf{SemdR}(l.\mathtt{rgt}) = \emptyset\end{array}}{[\![l]\!] = \mathsf{SID}}$$

$$\textsc{AO-Sid}_2\ \frac{\begin{array}{c}\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{lft}]\!] \notin \{\mathsf{UKD}, \mathsf{NPM}\} \quad [\![l.\mathtt{rgt}]\!] = \mathsf{RUD} \\ \mathsf{SemdR}(l.\mathtt{rgt}) \cap \mathsf{SemdR}(l.\mathtt{lft}) = \emptyset\end{array}}{[\![l]\!] = \mathsf{SID}}$$

$$\textsc{AO-Sid}_3\ \frac{\begin{array}{c}\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{lft}]\!] = [\![l.\mathtt{rgt}]\!] = \mathsf{RUD} \\ (\mathsf{dom}(l.\mathtt{lft}) \setminus \mathsf{SemdR}(l.\mathtt{rgt})) \cup (\mathsf{dom}(l.\mathtt{rgt}) \setminus \mathsf{SemdR}(l.\mathtt{lft})) \neq \emptyset\end{array}}{[\![l]\!] = \mathsf{SID}}$$

$$\textsc{Sid}\ \frac{\lambda_2(l) \in \{\oplus, \wedge, \vee\} \quad [\![l.\mathtt{rgt}]\!] = [\![l.\mathtt{lft}]\!] = \mathsf{SID} \quad \mathsf{SemdR}(l.\mathtt{lft}) \cap \mathsf{SemdR}(l.\mathtt{rgt}) = \emptyset}{[\![l]\!] = \mathsf{SID}}$$

$$\textsc{Not}\ \frac{\lambda_2(l) = \neg}{[\![l]\!] = [\![l.\mathtt{lft}]\!]} \qquad \textsc{No-Key}\ \frac{\mathsf{Supp}(l) \cap X_k = \emptyset}{[\![l]\!] = \mathsf{SID}} \qquad \textsc{Ukd}\ \frac{\text{No-rule is applicable at } l}{[\![l]\!] = \mathsf{UKD}}$$

Fig. 4. Semantic type-inference rules. The NPM type is not yet used here; its inference rules will be added in Figure 6, since they rely on the SMT solver-based analyses.

3.2 Checking Semantic Independence

Unlike $\mathsf{SuppR}(l)$, which only extracts syntactic information from the program and hence may be computed syntactically, $\mathsf{SemdR}(l)$ is more expensive to compute. In this subsection, we present a method that leverages the SMT solver to check, for any intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ and any random bit $r \in X_r$, whether c is semantically dependent of r. Specifically, we formulate it as a satisfiability (SAT) problem (formula Φ_s^r) defined as follows:

$$\Phi_s^r \equiv \Theta_s^{r=0}(c_0, X_p, X_k, X_r \setminus \{r\}) \wedge \Theta_s^{r=1}(c_1, X_p, X_k, X_r \setminus \{r\}) \wedge \Theta_s^{\neq}(c_0, c_1),$$

where $\Theta_s^{r=0}$ (respectively, $\Theta_s^{r=1}$) encodes the relation $\mathbf{I}(X_p, X_k, X_r)$ with r replaced by 0 (respectively, 1); c_0 and c_1 are copies of c and $\Theta_s^{\neq}$ asserts that the outputs differ even under the same inputs.

In logic synthesis and optimization, when $r \notin \mathsf{SemdR}(l)$, r will be called the *don't care* variable [52]. Therefore, it is easy to see why the following theorem holds.

Theorem 3.4. *Φ_s^r is unsatisfiable iff the value of r does not affect the value of c, i.e., c is semantically independent of r. Moreover, the formula size of Φ_s^r is linear in $|P|$.*

3.3 Verifying Higher-Order Masking

The type system so far targets *first-order* masking. We now outline how it extends to verify higher-order masking. Generally speaking, we have to check, for any d'-set $\{c_1, \ldots, c_{d'}\}$ of intermediate computation results such that $d' \leq d$, the joint distribution is either randomized to uniform distribution (RUD) or statistically secret independent (SID).

$$\textsc{Cp-Rud}\ \frac{\begin{array}{c}[\![c_1]\!] = \cdots = [\![c_n]\!] = \mathsf{RUD} \\ \exists r_1, \cdots, r_n \in X_r.\forall 1 \le i \ne j \le n.x_i \ne x_j \wedge x_i \in \mathsf{dom}(c_i)\end{array}}{[\![c_1, \cdots, c_n]\!] = \mathsf{RUD}}$$

$$\textsc{Cp-Sid}_1\ \frac{[\![c_1]\!] = \cdots = [\![c_n]\!] = \mathsf{SID} \qquad \forall 1 \le i \ne j \le n.\mathsf{SemdR}(c_i) \cap \mathsf{SemdR}(c_j) = \emptyset}{[\![c_1, \cdots, c_n]\!] = \mathsf{SID}}$$

$$\textsc{Cp-Sid}_2\ \frac{(\mathsf{Supp}(c_1) \cup \cdots \cup \mathsf{Supp}(c_n)) \cap X_k = \emptyset}{[\![c_1, \cdots, c_n]\!] = \mathsf{SID}}$$

$$\textsc{Cp-Sid}_3\ \frac{\begin{array}{c}[\![c_1, \cdots, c_n]\!] = \mathsf{RUD} \qquad [\![c'_1]\!] = \cdots = [\![c'_m]\!] = \mathsf{SID} \\ \forall 1 \le i \ne j \le m.\mathsf{SemdR}(c'_i) \cap \mathsf{SemdR}(c'_j) = \emptyset \\ \exists r_1, \cdots, r_n \in X_r.\forall 1 \le i \ne j \le n.x_i \ne x_j \wedge x_i \in \mathsf{dom}(c_i) \setminus \mathsf{SemdR}(c'_1, \cdots, c'_m)\end{array}}{[\![c_1, \cdots, c_n, c'_1, \cdots, c'_m]\!] = \mathsf{SID}}$$

$$\textsc{Cp-Ukd}\ \frac{\text{no-rule is applicable at } \{c_1, \cdots, c_k\}}{[\![c_1, \cdots, c_k]\!] = \mathsf{UKD}}$$

Fig. 5. Composition rules for handling *sets* of intermediate computation results, i.e., higher-order masking.

To tackle this problem, we lift SuppR and SemdR to *sets* of computation results as follows: for each d'-set $\{c_1, \ldots, c_{d'}\}$,

- $\mathsf{SuppR}(c_1, \ldots, c_{d'}) = \bigcup_{i \in [d']} \mathsf{SuppR}(c_i)$;
- $\mathsf{SemdR}(c_1, \ldots, c_{d'}) = \bigcup_{i \in [d']} \mathsf{SemdR}(c_i)$.

Our inference rules are extended by adding the composition rules shown in Figure 5.

Theorem 3.5. *For every d'-set $\{c_1, \ldots, c_{d'}\}$ of intermediate computation results,*

- *if $[\![c_1, \ldots, c_{d'}]\!] = \mathsf{RUD}$, then $\{c_1, \ldots, c_{d'}\}$ is guaranteed to be uniformly distributed, and hence perfectly masked;*
- *if $[\![c_1, \ldots, c_{d'}]\!] = \mathsf{SID}$, then $\{c_1, \ldots, c_{d'}\}$ is guaranteed to be perfectly masked.*

We remark that the SemdR function in these composition rules could also be safely replaced by the SuppR function, just as before.

4 THE GRADUAL REFINEMENT APPROACH FOR VERIFYING PERFECT MASKING

In this section, we present our method for gradually refining the type system by leveraging SMT solver-based techniques. Adding solvers to the sound type system makes it complete as well, thus allowing it to detect side-channel leaks whenever they exist, in addition to proving the absence of such leaks.

4.1 SMT-based Approach

For a given computation $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$, the verification of perfect masking (Definition 2.4) can be reduced to the *satisfiability* problem of the logical formula (Ψ) defined as follows:

$$\Psi \equiv \exists V_p.\exists V_k.\exists V'_k.\left(\sum_{V_r \in \{0,1\}^{|X_r|}} \mathbf{I}(V_p/X_p, V_k/X_k, V_r/X_r) \ne \sum_{V_r \in \{0,1\}^{|X_r|}} \mathbf{I}(V_p/X_p, V'_k/X'_k, V_r/X_r)\right).$$

Intuitively, given values (V_p, V_k) of (X_p, X_k), $count = \sum_{V_r \in \{0,1\}^{|X_r|}} \mathbf{I}(V_p/X_p, V_k/X_k, V_r/X_r)$ denotes the number of assignments of the random variable X_r under which $\mathbf{I}(V_p/X_p, V_k/X_k, V_r/X_r)$ is evaluated to logical 1. When random bits in X_r are uniformly distributed in the domain $\{0, 1\}$, $\frac{count}{2^{|X_r|}}$

is the probability of $\mathbf{I}(V_p/X_p, V_k/X_k, V_r/X_r)$ being logical 1 for the given pair (V_p, V_k). Therefore, Ψ is unsatisfiable if and only if c is perfectly masked.

Following Eldib et al. [39, 40], we encode the formula Ψ as a quantifier-free first-order logic formula Θ to be solved by an off-the-shelf SMT solver (e.g., Z3 [36]):

$$\Theta \equiv \left(\bigwedge_{V_r=0}^{2^{|X_r|}-1} \Theta_{X_k}^{V_r} \right) \wedge \left(\bigwedge_{V_r=0}^{2^{|X_r|}-1} \Theta_{X'_k}^{V_r} \right) \wedge \Theta_{b2i} \wedge \Theta_{\neq},$$

where

- $\Theta_{X_k}^{V_r}$ (respectively, $\Theta_{X'_k}^{V_r}$) for each $V_r \in \{0, \ldots, 2^{|X_r|} - 1\}$: encodes a copy of the input-output relation of $\mathbf{I}(X_p/X_p, X_k/X_k, V_r/X_r)$ (respectively, $\mathbf{I}(X_p/X_p, X'_k/X_k, V_r/X_r)$) by replacing X_r with concrete values V_r. There are $2^{|X_r|}$ distinct copies but share the same plaintext X_p.
- Θ_{b2i}: converts Boolean outputs of these copies to integers (true becomes 1 and false becomes 0) so that the number of assignments can be counted.
- $\Theta_{\neq}$: asserts the two summations, for X_k and X'_k, differ.

Example 4.1. For the example in Figure 2, verifying whether node c_4 is perfectly masked requires the SMT-based analysis. By instantiating (r_1, r_2) to values from $\{0, 1\}^2$, we can get the SMT encoding Θ, where the four components are given below:

$$\Theta_k \equiv \begin{pmatrix} (c_{41} = ((0 \oplus 0) \oplus (k \oplus 0)) \oplus (0 \oplus 0)) \wedge \\ (c_{42} = ((1 \oplus 0) \oplus (k \oplus 0)) \oplus (1 \oplus 0)) \wedge \\ (c_{43} = ((0 \oplus 1) \oplus (k \oplus 1)) \oplus (0 \oplus 1)) \wedge \\ (c_{44} = ((1 \oplus 1) \oplus (k \oplus 1)) \oplus (1 \oplus 1)) \end{pmatrix}$$

$$\Theta_{k'} \equiv \begin{pmatrix} (c'_{41} = ((0 \oplus 0) \oplus (k' \oplus 0)) \oplus (0 \oplus 0)) \wedge \\ (c'_{42} = ((1 \oplus 0) \oplus (k' \oplus 0)) \oplus (1 \oplus 0)) \wedge \\ (c'_{43} = ((0 \oplus 1) \oplus (k' \oplus 1)) \oplus (0 \oplus 1)) \wedge \\ (c'_{44} = ((1 \oplus 1) \oplus (k' \oplus 1)) \oplus (1 \oplus 1)) \end{pmatrix}$$

$$\Theta_{b2i} \equiv \begin{pmatrix} (((n_1 = 1) \wedge c_{41}) \vee ((n_1 = 0) \wedge \neg c_{41})) \wedge \\ (((n_2 = 1) \wedge c_{42}) \vee ((n_2 = 0) \wedge \neg c_{42})) \wedge \\ (((n_3 = 1) \wedge c_{43}) \vee ((n_3 = 0) \wedge \neg c_{43})) \wedge \\ (((n_4 = 1) \wedge c_{44}) \vee ((n_4 = 0) \wedge \neg c_{44})) \wedge \\ (((n'_1 = 1) \wedge c'_{41}) \vee ((n'_1 = 0) \wedge \neg c'_{41})) \wedge \\ (((n'_2 = 1) \wedge c'_{42}) \vee ((n'_2 = 0) \wedge \neg c'_{42})) \wedge \\ (((n'_3 = 1) \wedge c'_{43}) \vee ((n'_3 = 0) \wedge \neg c'_{43})) \wedge \\ (((n'_4 = 1) \wedge c'_{44}) \vee ((n'_4 = 0) \wedge \neg c'_{44})) \end{pmatrix}$$

$$\Theta_{\neq} \equiv (n_1 + n_2 + n_3 + n_4) \neq (n'_1 + n'_2 + n'_3 + n'_4)$$

We convert Boolean to integer by adding predicates of the form $((n = 1) \wedge c) \vee ((n = 0) \wedge \neg c)$, which ensures that if the Boolean variable c is true, then the integer n must be 1; otherwise, n must be 0.

By invoking the SMT solver six times, one can get the following result: $[\![c_1]\!] = [\![c_2]\!] = [\![c_3]\!] = [\![c_4]\!] = [\![c_5]\!] = [\![c_6]\!] = \texttt{SID}$.

Although the SMT formula size is linear in $|P|$, the number of distinct copies is exponential of the number of random bits used in the computation. Thus, the approach cannot be applied to large programs. To overcome the problem, incremental algorithms [39, 40] were proposed to reduce the formula size using partitioning and heuristic reduction.

Incremental SMT-based approach. Given a computation $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ that corresponds to a subtree T rooted at l in the DDG, we search for an internal node l_s in T (a *cut-point*) such that

$$\text{AO-NPM}_1 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{rgt}]\!] = \texttt{NPM} \quad [\![l.\texttt{lft}]\!] = \texttt{RUD} \quad \texttt{dom}(l.\texttt{lft}) \setminus \texttt{SemdR}(l.\texttt{rgt}) \neq \emptyset}{[\![l]\!] = \texttt{NPM}}$$

$$\text{AO-NPM}_2 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{lft}]\!] = \texttt{NPM} \quad [\![l.\texttt{rgt}]\!] = \texttt{RUD} \quad \texttt{dom}(l.\texttt{rgt}) \setminus \texttt{SemdR}(l.\texttt{lft}) \neq \emptyset}{[\![l]\!] = \texttt{NPM}}$$

$$\text{CP-NPM} \frac{[\![c_{k+1}]\!] = \texttt{NPM}}{[\![c_1, \cdots, c_{k+1}]\!] = \texttt{NPM}}$$

Fig. 6. Complementary rules used during refinement of the type inference (Figure 4).

$\texttt{dom}(l_s) \cap \texttt{unq}(l) \neq \emptyset$. A cut-point is *maximal* if there is no other cut-point from l to l_s. Let $\widehat{T}$ be the *simplified tree* obtained from T by replacing every subtree rooted at a maximal cut-point with a random variable from $\texttt{dom}(l_s) \cap \texttt{unq}(l)$. Then, $[\![c]\!]$ in $\widehat{T}$ is SID iff $[\![c]\!]$ in T is SID.

The main observation is that if l_s is a cut-point, then there is a random variable $r \in \texttt{dom}(l_s) \cap \texttt{unq}(l)$, which implies $[\![l_s]\!]$ is RUD. Here, $r \in \texttt{unq}(l)$ implies $\lambda_1(l_s)$ can be seen as a *fresh* random variable when we evaluate l. Consider the node c_3 in the example in Figure 2, it is easy to see $r_1 \in \texttt{dom}(c_2) \cap \texttt{unq}(c_3)$. Therefore, for the purpose of verifying c_3, the entire subtree rooted at c_2 can be replaced by the random variable r_1.

In addition to partitioning, heuristics rules [39, 40] can be used to simplify or avoid the SMT solving. (1) When constructing formula Θ of c, all random variables in $\texttt{SuppR}(l) \setminus \texttt{SemdR}(l)$, which are *don't cares*, can be replaced by constant 1 or 0. (2) The No-Key and Sid rules in Figure 4 with the SuppR function are used to skip some checks by SMT solving in References [39, 40].

Example 4.2. When applying incremental SMT-based approach to the example in Figure 2, c_1 has to be decided by SMT solving, but c_2 is skipped due to No-Key rule.

As for c_3, since $r_1 \in \texttt{dom}(c_2) \cap \texttt{unq}(c_3)$, c_2 is a cut-point and the subtree rooted at c_2 can be replaced by r_1, leading to the simplified computation $r_1 \oplus (r_2 \oplus k)$—subsequently, it is skipped by the Sid rule with SuppR. Note that the above Sid rule is not applicable to the original subtree, because r_2 occurs in the support of both children of c_3.

There is no cut-point for c_4, so it is checked using the SMT solver. But since c_4 is semantically independent of r_1 (a *don't care* variable), to reduce the SMT formula size, we replace r_1 by 1 (or 0) when constructing the SMT formula Θ.

4.2 Feeding SMT-based Analysis Results Back to Type System

Consider a scenario where initially the type system (cf. Section 3) failed to resolve a node l, i.e., $[\![l]\!] = \texttt{UKD}$, but the SMT-based approach resolved it as either NPM or SID. Such results should be *fed back* to improve the type system, which may lead to the following two favorable outcomes: (1) marking more nodes as perfectly masked (RUD or SID) and (2) marking more nodes as leaky (NPM), which means we can avoid expensive SMT calls for these nodes. More specifically, if SMT-based analysis shows that l is perfectly masked, then the type of l can be refined to $[\![l]\!] = \texttt{SID}$; feeding it back to the type system allows us to infer more types for nodes that syntactically depend on l.

However, if SMT-based analysis shows l is not perfectly masked, then the type of l can be refined to $[\![l]\!] = \texttt{NPM}$; feeding it back allows the type system to infer that other nodes may be NPM as well. To achieve what is outlined in the second case above, we add the NPM-related type inference rules shown in Figure 6. When they are added to the type system outlined in Figure 4, more NPM-typed nodes will be deduced, which allows our method to skip the (more expensive) checking of some nodes using the SMT-based analysis.

```
1  bool  compute(bool k1, bool r1, bool r2,
2                bool r3, bool r4, bool r5){
3    bool c1, c2, c3, c4, c5, c6, c7;
4    c1 = k1 ⊕ r1;
5    c2 = c1 ∧ r2;
6    c3 = r1 ∧ r2;
7    c4 = c3 ⊕ r3;
8    c5 = c2 ∨ c4;
9    c6 = r4 ∧ r5;
10   c7 = c5 ⊕ c6;
11   return c7;
12 }
```

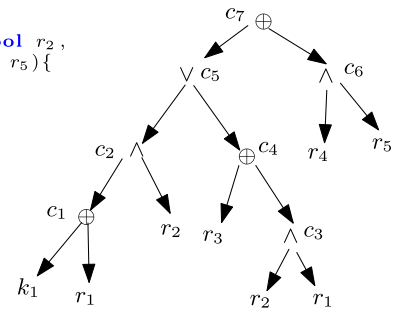

Fig. 7. An example for feeding back: left-hand part is the C-like program and right-hand part is its data dependency graph, where $r_1 - r_5$ are random variables, and k_1 is a secret variable.

ALGORITHM 1: Procedure SCInfer(P, X_p, X_k, X_r, π)

Procedure SCInfer(P, X_p, X_k, X_r, π)
  foreach $l \in N$ *in a topological order from leaf to root* **do**
    if l *is a leaf* **then** $\pi(l) := [\![l]\!]$;
    else
      TypeInfer$(l, P, X_p, X_k, X_r, \pi, \mathsf{SuppR})$;
      if $\pi(l) = \mathsf{UKD}$ **then**
        let $\widehat{P}$ be the simplified tree of the subtree rooted at l in P;
        TypeInfer$(l, \widehat{P}, X_p, X_k, X_r, \pi, \mathsf{SemdR})$;
        if $\pi(l) = \mathsf{UKD}$ **then**
          res:=CheckBySMT$(\widehat{P}, X_p, X_k, X_r)$;
          if res=Not-Perfectly-Masked **then**
            $\pi(l) := \mathsf{NPM}$;
          else if res=Perfectly-Masked **then**
            $\pi(l) := \mathsf{SID}$;
          else $\pi(l) := \mathsf{UKD}$;

Example 4.3. Consider the program in Figure 7; by applying the original type inference approach with either SuppR or SemdR, we have

$[\![c_1]\!] = [\![c_4]\!] = \mathsf{RUD}$, $[\![c_2]\!] = [\![c_3]\!] = [\![c_6]\!] = \mathsf{SID}$, $[\![c_5]\!] = [\![c_7]\!] = \mathsf{UKD}$.

In contrast, by applying SMT-based analysis to c_5, we can deduce $[\![c_5]\!] = \mathsf{SID}$. Feeding $[\![c_5]\!] = \mathsf{SID}$ back to the original type system, and then applying the Sid rule to $c_7 = c_5 \oplus c_6$, we are able to deduce $[\![c_7]\!] = \mathsf{SID}$. Without refinement, this was not possible.

4.3 The Overall Algorithm for Verifying Perfect Masking

Having presented all the components, we now present the overall procedure for verifying perfect masking, which integrates the semantic type system and SMT-based method for gradual refinement. Algorithm 1 shows the pseudo code. Given the program P, the sets of public (X_p), secret (X_k), random (X_r) variables and an empty map π, it invokes SCInfer(P, X_p, X_k, X_r, π) to traverse the DDG in a topological order and annotate every node l with a distribution type from T. The subroutine TypeInfer implements the type inference rules outlined in Figure 4 and Figure 6, where the parameter f can be either SuppR or SemdR.

SCInfer first deduces the type of each node $l \in N$ by invoking TypeInfer with $f = \mathsf{SuppR}$. Once a node l is annotated as UKD, a simplified subtree $\widehat{P}$ of the subtree rooted at l is constructed.

ALGORITHM 2: Procedure TypeInfer$(l, P, X_p, X_k, X_r, \pi, f)$

Procedure TypeInfer$(l, P, X_p, X_k, X_r, \pi, f)$
  if $\lambda_2(l) = \neg$ **then** $\pi(l) := \pi(l.\texttt{lft})$;
  else if $\lambda_2(l) = \oplus$ **then**
    if $\pi(l.\texttt{lft}) = \texttt{RUD} \wedge \texttt{dom}(l.\texttt{lft}) \setminus f(l.\texttt{rgt}) \neq \emptyset$ **then**
      $\pi(l) := \texttt{RUD}$;
    else if $\pi(l.\texttt{rgt}) = \texttt{RUD} \wedge \texttt{dom}(l.\texttt{rgt}) \setminus f(l.\texttt{lft}) \neq \emptyset$ **then**
      $\pi(l) := \texttt{RUD}$;
    else if $\pi(l.\texttt{rgt}) = \pi(l.\texttt{lft}) = \texttt{SID} \wedge f(l.\texttt{lft}) \cap f(l.\texttt{rgt}) \cap X_r = \emptyset$ **then**
      $\pi(l) := \texttt{SID}$;
    else if $\texttt{Supp}(l) \cap X_k = \emptyset$ **then**
      $\pi(l) := \texttt{SID}$;
    else $\pi(l) := \texttt{UKD}$;
  else
    if $\left(\begin{array}{c}\Big((\pi(l.\texttt{lft}) = \texttt{RUD} \wedge \pi(l.\texttt{rgt}) \notin \{\texttt{UKD}, \texttt{NPM}\}) \vee \\ (\pi(l.\texttt{rgt}) = \texttt{RUD} \wedge \pi(l.\texttt{lft}) \notin \{\texttt{UKD}, \texttt{NPM}\})\Big) \wedge \\ f(l.\texttt{lft}) \cap f(l.\texttt{rgt}) \cap X_r = \emptyset\end{array}\right)$ **then**
      $\pi(l) := \texttt{SID}$;
    else if $\left(\begin{array}{c}(\texttt{dom}(l.\texttt{rgt}) \setminus f(l.\texttt{lft})) \cup (\texttt{dom}(l.\texttt{lft}) \setminus f(l.\texttt{rgt})) \neq \emptyset \\ \wedge \pi(l.\texttt{lft}) = \texttt{RUD} \wedge \pi(l.\texttt{rgt}) = \texttt{RUD}\end{array}\right)$ **then**
      $\pi(l) := \texttt{SID}$;
    else if $\left(\begin{array}{c}(\pi(l.\texttt{lft}) = \texttt{RUD} \wedge \pi(l.\texttt{rgt}) = \texttt{NPM} \wedge \texttt{dom}(l.\texttt{lft}) \setminus f(l.\texttt{rgt}) \neq \emptyset) \vee \\ (\pi(l.\texttt{rgt}) = \texttt{RUD} \wedge \pi(l.\texttt{lft}) = \texttt{NPM} \wedge \texttt{dom}(l.\texttt{rgt}) \setminus f(l.\texttt{lft}) \neq \emptyset)\end{array}\right)$ **then**
      $\pi(l) := \texttt{NPM}$;
    else if $(\pi(l.\texttt{lft}) = \pi(l.\texttt{rgt}) = \texttt{SID}) \wedge f(l.\texttt{lft}) \cap f(l.\texttt{rgt}) \cap X_r = \emptyset$ **then**
      $\pi(l) := \texttt{SID}$;
    else if $\texttt{SuppR}(l) \cap X_k = \emptyset$ **then**
      $\pi(l) := \texttt{SID}$;
    else $\pi(l) := \texttt{UKD}$;

Next, TypeInfer with $f = \texttt{SemdR}$ is invoked to resolve the UKD node in $\widehat{P}$. If $\pi(l)$ becomes non-UKD afterward, then TypeInfer with $f = \texttt{SuppR}$ is invoked again to quickly deduce the types of the fan-out nodes in P. But if $\pi(l)$ remains UKD, then SCInfer invokes the incremental SMT-based approach to decide whether l is either SID or NPM. This is sound and complete, unless the SMT solver runs out of time/memory, in which case UKD is assigned to l.

Theorem 4.4. *For every intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ corresponding to a subtree rooted at l, our method in* QMSInfer *guarantees to return sound and complete results:*

- $\pi(l) = \texttt{RUD}$ *iff c is uniformly distributed, and hence perfectly masked;*
- $\pi(l) = \texttt{SID}$ *iff c is statistically secret independent, i.e., perfectly masked;*
- $\pi(l) = \texttt{NPM}$ *iff c is not perfectly masked (leaky);*

If timeout or memory out is used to bound the execution of the SMT solver, then it is also possible that $\pi(l) = \texttt{UKD}$, meaning c has an unknown distribution (it may or may not be perfectly masked). It is interesting to note that if we regard UKD as *potential leak* and at the same time bound (or even disable) SMT-based analysis, then Algorithm 1 degenerates to a *sound* type system that is both fast and potentially accurate.

5 THE GRADUAL REFINEMENT APPROACH FOR COMPUTING QMS VALUES

In this section, we present our approach for computing QMS values. We first recall the SMT-based approach for checking a QMS requirement for each intermediate computation result from References [41, 42]. Then, we propose a binary search-based approach to compute the accurate QMS value of each intermediate computation result.

5.1 Checking a QMS Requirement

The SMT-based approach for checking a QMS requirement is a generalization of the one for checking perfect masking. Given an intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ and a QMS requirement q, we reduce the problem of checking $\texttt{QMS}_{\mathbf{I}} \geq q$ to the satisfiability problem of a (quantifier-free) first-order logic formula. Recall that $\texttt{QMS}_{\mathbf{I}} = 1 - \max_{V_p, V_p', V_k'}(E(\mathbf{I}(V_p/X_p, V_k/X_k)) - E(\mathbf{I}(V_p/X_p, V_k'/X_k)))$. To check whether $\texttt{QMS}_{\mathbf{I}} \geq q$, it suffices to check the unsatisfiability of the following formula:

$$\exists V_p, V_k, V_k'.\Big(\sharp(\mathbf{I}(V_p/X_p, V_k/X_k)) - \sharp(\mathbf{I}(V_p/X_p, V_k'/X_k))\Big) > \Delta_{\mathbf{I}}^q,$$

where $\sharp(\mathbf{I}(V_p/X_p, V_k/X_k))$ and $\sharp(\mathbf{I}(V_p/X_p, V_k'/X_k))$ denote the number of satisfying assignments of $\mathbf{I}(V_p/X_p, V_k/X_k)$ and $\mathbf{I}(V_p/X_p, V_k'/X_k)$, respectively, and $\Delta_{\mathbf{I}}^q = (1-q) \times 2^{|X_r|}$.

We encode it as a logic formula $\Psi_{\mathbf{I}}^q$ to be solved by an off-the-shelf SMT solver (e.g., Z3):

$$\Psi_{\mathbf{I}}^q \equiv \left(\bigwedge_{V_r=0}^{2^{|X_r|}-1} \Theta_{X_k}^{V_r}\right) \wedge \left(\bigwedge_{V_r=0}^{2^{|X_r|}-1} \Theta_{X_k'}^{V_r}\right) \wedge \Theta_{\texttt{b2i}} \wedge \Theta_{\texttt{diff}}^q,$$

where

- $\Theta_{X_k}^{V_r}$ (respectively, $\Theta_{X_k'}^{V_r}$) for $V_r \in \{0, \ldots, 2^{|X_r|}-1\}$: encodes a copy of the input-output relation of $\mathbf{I}(X_k/X_k, V_r/X_r)$ (respectively, $\mathbf{I}(X_k'/X_k, V_r/X_r)$) by replacing X_r with concrete values V_r and variable X_k with X_k (respectively, X_k'). There are $2^{|X_r|}$ distinct copies, but share the same plaintext X_p.
- $\Theta_{\texttt{b2i}}$: converts Boolean outputs of these copies to integers (true becomes 1 and false becomes 0) so that the number of assignments can be counted.
- $\Theta_{\texttt{diff}}^q$: asserts the difference of the two sums for X_k and X_k' is larger than $\Delta_{\mathbf{I}}^q$.

Theorem 5.1. *$\Psi_{\mathbf{I}}^q$ is unsatisfiable iff $\texttt{QMS}_{\mathbf{I}} \geq q$, and the size of $\Psi_{\mathbf{I}}^q$ is polynomial in $|P|$ and exponential in number of random bits in the intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$.*

Example 5.2. Consider the node $n_7 = r_2 \wedge (k_1 \oplus r_1)$ of the example program in Figure 3, the SMT encoding $\Psi_{n_7}^{0.5}$ is given as follows:

$$\begin{array}{c}
\left(\begin{array}{l}(b_{00} = (0 \wedge (k_1 \oplus 0))) \wedge (b_{01} = (0 \wedge (k_1 \oplus 1)))\wedge \\ (b_{10} = (1 \wedge (k_1 \oplus 0))) \wedge (b_{11} = 1(\wedge(k_1 \oplus 1)))\wedge\end{array}\right) \wedge \\
\left(\begin{array}{l}(b_{00}' = (0 \wedge (k_1' \oplus 0))) \wedge (b_{01}' = (0 \wedge (k_1' \oplus 1)))\wedge \\ (b_{10}' = (1 \wedge (k_1' \oplus 0))) \wedge (b_{11}' = (1 \wedge (k_1' \oplus 1)))\end{array}\right) \wedge \\
\left(\begin{array}{l}(d_1 = (b_{00} \ ?\ 1 : 0)) \wedge (d_2 = (b_{01} \ ?\ 1 : 0))\wedge \\ (d_3 = (b_{10} \ ?\ 1 : 0)) \wedge (d_4 = (b_{11} \ ?\ 1 : 0))\wedge \\ (d_1' = (b_{00}' \ ?\ 1 : 0)) \wedge (d_2' = (b_{01}' \ ?\ 1 : 0))\wedge \\ (d_3' = (b_{10}' \ ?\ 1 : 0)) \wedge (d_4' = (b_{11}' \ ?\ 1 : 0))\end{array}\right) \wedge \\
(d_1 + d_2 + d_3 + d_4) - (d_1' + d_2' + d_3' + d_4') > 0.5 * 2^2.
\end{array}$$

Based on the above SMT encoding, we present an algorithm for checking a program against a QMS requirement, which is more general than verifying perfect masking.

ALGORITHM 3: Procedure CheckQMS(P, X_p, X_k, X_r, q)

```
Procedure CheckQMS(P, X_p, X_k, X_r, q)
    SCInfer(P, X_p, X_k, X_r, π);
    forall the l ∈ N such that π(l) = NPM and I(X_p, X_k, X_r) is its computation do
        if SMTSolver(Ψ_I^q) =SAT then
            return False;
    return True;
```

ALGORITHM 4: Procedure QMSInfer(P, X_p, X_k, X_r)

```
Procedure QMSInfer(P, X_p, X_k, X_r)
    SCInfer(P, X_p, X_k, X_r, π);
    foreach l ∈ N with I(X_p, X_k, X_r) being corresponding computation do
        if π(l) ∈ {SID, RUD, CST} then QMS_I := 1;
        else
            if SemdR(l) = ∅ then
                QMS_I := 0;
            else
                low := 0;
                high := 2^|SemdR(l)|;
                while low < high do
                    mid := ⌈(low+high)/2⌉;
                    q := mid / 2^|SemdR(l)|;
                    if SMTSolver(Ψ_I^q) =SAT then
                        high := mid − 1;
                    else low := mid;
                QMS_I := low / 2^|SemdR(l)|;
```

Given a program P, the set of public (X_p), secret (X_k), random (X_r) variables and a QMS requirement q, CheckQMS in Algorithm 3 returns `True` if $\mathtt{QMS}_\mathbf{I} \geq q$ holds for every intermediate computation result $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ and returns `False` otherwise. Inside CheckQMS, it first invokes SCInfer to check whether $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ is perfectly masked. If it is perfectly masked, then the corresponding $\mathtt{QMS}_\mathbf{I}$ is set to 1 directly. Otherwise, for each $c \leftarrow \mathbf{I}(X_p, X_k, X_r)$ that is not perfectly masked, the SMT-based approach is used to check whether $\mathtt{QMS}_\mathbf{I} \geq q$.

5.2 Computing the QMS

In this subsection, we present our algorithm for computing the QMS value of an intermediate computation result instead of checking for a fixed QMS requirement. Given the program P and the set of public (X_p), secret (X_k), and random (X_r) variables, QMSInfer first invokes SCInfer to check perfect masking. For each l with intermediate computation result $\mathbf{I}$, if it is perfectly masked, we can directly get that $\mathtt{QMS}_\mathbf{I}$ is 1. Otherwise, we first check whether $\mathtt{SemdR}(l)$ is empty or not. If $\mathtt{SemdR}(l)$ is empty, then we can conclude that $\mathtt{QMS}_\mathbf{I} = 0$. Otherwise, we use a binary search to compute $\mathtt{QMS}_\mathbf{I}$ based on the following observation:

$$\mathtt{QMS}_\mathbf{I} \in \left\{ \frac{i}{2^{|\mathtt{SemdR}(l)|}} : 0 \leq i \leq 2^{|\mathtt{SemdR}(l)|} \right\}.$$

Algorithm 4 shows the pseudo code for computing the QMS values of all nodes in the DDG. Note that the while-loop executes at most $\mathbf{O}(|\mathtt{SemdR}(l)|)$ times for each node l.

ALGORITHM 5: Procedure MinQMSInfer(P, X_p, X_k, X_r)

```
Procedure MinQMSInfer(P, X_p, X_k, X_r)
    minQMS := 1;
    SCInfer(P, X_p, X_k, X_r, π);
    foreach l ∈ N with I(X_p, X_k, X_r) being corresponding computation do
        if π(l) ∉ {SID, RUD, CST} then
            if SemdR(l) = ∅ then
                return 0;
            else
                low := 0;
                high := ⌈minQMS × 2^|SemdR(l)|⌉;
                while low < high do
                    mid := ⌈(low+high)/2⌉;
                    q := mid / 2^|SemdR(l)|;
                    if SMTSolver(Ψ_I^q) =SAT then
                        high := mid − 1;
                    else low := mid;
                minQMS := min(minQMS, low / 2^|SemdR(l)|);
                if minQMS = 0 then
                    return minQMS;
    return minQMS;
```

Our algorithm for computing QMS values is different from the one proposed by Eldib et al. [41, 42]. Their algorithm computes QMS values by directly searching for the QMS requirement q between 0 to 1 with step 0.01. Hence, it computes only an approximation of the QMS value and the search iterates at most 10 times for each intermediate computation result. In contrast, our approach takes into account the number of random variables in the intermediate computation result, and it computes the accurate QMS values. Furthermore, our approach more efficient when the number of random variables in intermediate computation results is small.

QMSInfer is able to compute the QMS values of all the intermediate computation results, which quantify the amount of information leakage through the side channel. In practice, one may be more interested in computing the minimal QMS value of all the intermediate computation results, which can be regarded as the weakest part of the masking countermeasure. Although one can compute the minimal QMS value by first computing all the QMS values using QMSInfer, we propose a more efficient algorithm, shown in Algorithm 5.

Algorithm 5 is a modification of Algorithm 4. Given the program P and the set of public (X_p), secret (X_k), and random (X_r) variables, MinQMSInfer first initializes the variable minQMS as 1, which will be updated if a smaller QMS value is obtained. MinQMSInfer then invokes SCInfer to check perfect masking. Next, for each leaky node l with the corresponding intermediate computation result $\mathbf{I}$, it uses a binary search to compute $\mathtt{QMS}_{\mathbf{I}}$ with upper bound $\lceil \mathtt{minQMS} \times 2^{|\mathtt{SemdR}(l)|} \rceil$, instead of $2^{|\mathtt{SemdR}(l)|}$, as it suffices to consider QMS requirements that are smaller than the current minimal QMS value minQMS.

6 EXPERIMENTS

We have implemented our method in a verification tool named QMSInfer, which uses Z3 [36] as the underlying SMT solver. We also implemented the syntactic type inference approach [65], the incremental SMT-based verification approach [39, 40], and the SMT-based QMS computing

Table 1. Benchmark Statistics, Where Column *Name* Denotes the Name of the Benchmark, Column *Description* Gives a Briefly Description of the Benchmark, Column ♯*Loc* Presents the Number of Locations in the Benchmark, Column ♯*Nodes* Denotes the Number of Internal Nodes in DDG, and Columns $|X_k|$, $|X_p|$, and $|X_r|$, Respectively, Give the Number of Secret Variables, Public Variables, and Random Variables in the Benchmark

Name	Description	♯Loc	♯Nodes	$\|X_k\|$	$\|X_p\|$	$\|X_r\|$
P1	CHES13 Masked Key Whitening	79	32	16	16	16
P2	CHES13 De-mask and then Mask	67	38	8	0	16
P3	CHES13 AES Shift Rows	21	6	2	0	2
P4	CHES13 Messerges Boolean to Arithmetic (bit0)	23	6	2	0	2
P5	CHES13 Goubin Boolean to Arithmetic (bit0)	27	8	1	0	2
P6	Logic Design for AES S-Box (1st implementation)	32	9	2	0	2
P7	Logic Design for AES S-Box (2nd implementation)	40	11	2	0	3
P8	Masked Chi function MAC-Keccak (1st implementation)	59	18	3	0	4
P9	Masked Chi function MAC-Keccak (2nd implementation)	60	18	3	0	4
P10	Syn. Masked Chi func MAC-Keccak (1st implementation)	66	28	3	0	4
P11	Syn. Masked Chi func MAC-Keccak (2nd implementation)	66	28	3	0	4
P12	MAC-Keccak 512b Perfect masked	426k	197k	288	288	3205
P13	MAC-Keccak 512b De-mask and then mask (compiler error)	426k	197k	288	288	3205
P14	MAC-Keccak 512b Not-perfect Masking of Chi function (v1)	426k	197k	288	288	3205
P15	MAC-Keccak 512b Not-perfect Masking of Chi function (v2)	429k	198k	288	288	3205
P16	MAC-Keccak 512b Not-perfect Masking of Chi function (v3)	426k	197k	288	288	3205
P17	MAC-Keccak 512b Unmasking of Pi function	442k	205k	288	288	3205

approach [41, 42] in the same tool for experimental comparison purposes. We conducted experiments on publicly available cryptographic software implementations, including fragments of AES and MAC-Keccak [39, 40]. Our experiments were conducted on a machine with 64-bit Ubuntu 12.04 LTS, Intel Xeon(R) CPU E5-2603 v4, and 32GB RAM.

Overall, results of our experiments show that

- QMSInfer is significantly more accurate than prior syntactic type inference approach [65] for checking perfect masking; indeed, it solved thousands of UKD cases reported by the prior technique;
- QMSInfer is almost twice faster than prior SMT-based approach [39, 40] for checking perfect masking on the large programs while maintaining the same accuracy; for example, QMSInfer verified the benchmark named P12 in a few seconds, whereas the prior SMT-based method took more than an hour.
- QMSInfer is significantly more accurate and faster than prior SMT-based approach for computing the QMS values [41, 42].

6.1 Benchmarks

Table 1 shows the detailed statistics of the benchmarks, including 17 examples (P1–P17), all of which have nonlinear operations. Columns 1 and 2 show the name of the program and a short description. Column 3 shows the number of instructions in the probabilistic Boolean program. Column 4 shows the number of internal nodes in DDG denoting intermediate computation results. The remaining columns show the number of bits in the secret, public, and random variables, respectively. Remark that the number of random variables in each intermediate computation result is far less than the one of the program. All these programs are transformed into Boolean programs

Table 2. Experimental Results: Comparison of Three Perfect Masking Verification Approaches, Where Column *Masked* Gives the Ground Truth (*Yes* Denoting Perfectly Masked, Otherwise *No*), Column ♯UKD Gives the Number of UKD-typed Nodes, Column ♯NPM Gives the Number of NPM-typed Nodes, and Column ♯*By SMT* Denotes the Number of Nodes That Are Checked by Invoking the SMT-based Approach

Name	Masked	Syn. Infer [65]		SMT App [39, 40]			QMSInfer		
		♯UKD	Time	♯NPM	♯By SMT	Time	♯NPM	♯By SMT	Time
P1	No	16	≈0s	16	16	0.39s	16	16	0.41s
P2	No	8	≈0s	8	8	0.28s	8	8	0.73s
P3	Yes	0	≈0s	0	0	≈0s	0	0	0s
P4	Yes	**3**	≈0s	0	**3**	0.16s	**0**	**0**	0.13s
P5	Yes	**3**	≈0s	0	**3**	0.15s	**0**	**2**	0.36s
P6	No	2	≈0s	2	2	0.11s	2	2	0.27s
P7	No	2	0.01s	1	2	0.11s	1	1	0.20s
P8	No	3	≈0s	3	3	0.15s	3	3	0.31s
P9	No	2	≈0s	2	2	0.11s	2	2	0.27s
P10	No	3	≈0s	1	2	0.15s	1	2	0.28s
P11	No	4	≈0s	1	3	0.2s	1	3	0.37s
P12	Yes	0	**1m 5s**	0	0	92m 8s	0	0	**4.44s**
P13	No	4800	1m 11s	4800	4800	95m 30s	4800	4800	47m 16s
P14	No	3200	1m 11s	3200	3200	118m 1s	3200	3200	55m 25s
P15	No	**3200**	1m 21s	1600	3200	127m 45s	**1600**	**3200**	58m 35s
P16	No	4800	1m 13s	4800	4800	123m 54s	4800	4800	63m 26s
P17	No	17600	1m 14s	17600	16000	336m 51s	**17600**	**12800**	109m 16s

where each instruction has at most two operands. Since the statistics were collected from the transformed code, they may have minor differences from statistics reported in prior work [39, 40].

In particular, P1–P5 are masking examples originated from Reference [15], P6 and P7 are originated from Reference [22], P8 and P9 are the MAC-Keccak computation reordered examples originated from Reference [16], and P10 and P11 are two experimental masking schemes for the Chi function in MAC-Keccak. Among the larger programs, P12–P17 are the regenerations of MAC-Keccak reference code submitted to the SHA-3 competition held by NIST, where P13–P16 implement the masking of Chi functions using different masking schemes and P17 implements the de-masking of Pi function.

6.2 Experimental Results on Verifying Perfect Masking

We compare the performance of QMSInfer, the purely syntactic type inference method (denoted Syn. Infer) and the incremental SMT-based method (denoted by SMT App). Table 2 shows the results. Column 1 shows the name of each benchmark. Column 2 shows whether it is perfectly masked (ground truth). Columns 3 and 4 show the results of the purely syntactic type inference method, including the number of nodes inferred as UKD type and the time. Columns 5–7 (respectively, Columns 8–10) show the results of the incremental SMT-based method (respectively, our method QMSInfer), including the number of leaky nodes (NPM type), the number of nodes actually checked by the SMT-based approach, and the time.

Compared with syntactic type inference method, our approach is significantly more accurate (e.g., see P4, P5, and P15), where many of UKD-typed nodes are refined to either NPM type or SID type. Furthermore, the time taken by both methods are comparable on small programs. On the large

programs that are not perfectly masked (i.e., P13–P17), our method is slower since QMSInfer has to resolve the UKD nodes reported by syntactic inference. However, it is interesting to note that, on the perfectly masked large program (P12), our method is faster. Indeed, every intermediate computation result in P12 is syntactically masked by a unique random variable that allow us to prove it using the syntactic type inference system. Masking each intermediate computation result by a unique random variable is a possible solution, but not efficient in practice, as generation of random values is very time-consuming.

Moreover, the UKD-typed nodes in P4, reported by the purely syntactic type inference method, are all proved to be perfectly masked by our semantic type inference system, without calling the SMT solver at all. As for the three UKD-typed nodes in P5, our method proves them all by invoking the SMT solver only twice; it means that the feedback of the new SID types (discovered by SMT-based approach) allows our type system to improve its accuracy, which turns the third UKD-type node to SID type.

Finally, compared with the original SMT-based approach, our method is almost twice faster on the large programs (e.g., P12–P17). Furthermore, the number of nodes actually checked by invoking the SMT solver is also lower than in the original SMT-based approach (e.g., P4 and P5, and P17). In particular, there are 3,200 UKD-typed nodes in P17, which are refined into NPM type by our new inference rules (cf. Figure 6), and thus avoid the more expensive SMT calls.

To summarize, results of our experiments show that QMSInfer is fast in obtaining proofs in perfectly masked programs, while retaining the ability to detect real leaks in not perfectly masked programs and is scalable for handling realistic applications.

Detailed Statistics. Table 3 and Table 4 show more detailed statistics of our approach on verifying perfect masking.

In Table 3, Columns 2–5 show the number of nodes in each distribution type deduced by our method. Column 6 and Column 7 show the number of UKD-typed nodes that are proved by the SMT-based approach and the semantic type inference, respectively. Column 8 and Column 9 show the number of UKD-typed nodes that are refined to NPM type and SID type, respectively.

Results in Table 3 indicate that most of the DDG nodes in these benchmark programs are either RUD or SID, and almost all of them can be quickly deduced by our type system. Column 4 shows that, at least in these benchmark programs, Boolean constants are rare. Column 6 and Column 7 indicate that most of UKD-typed nodes are resolved by the SMT-based approach, and the semantic type system works in some cases. Columns 8 and 9 show that if our refined type system fails to prove perfect masking, it is usually not perfectly masked. One may notice that type inference with SemdR does not make sense on benchmarks P12-P17. We argue that all of P12-P17 are the regenerations of MAC-Keccak reference code. We plan to analyze more benchmarks in future.

In Table 4, Column 2 shows the number of SMT calls for computing SemdR, and Column 3 shows the corresponding time. Column 4 shows the number of SMT calls for checking *don't care* variables used to reduce SMT formula size, and Column 5 shows the corresponding time for computing all the *don't care* variables. Column 6 shows the number of SMT calls used by SMT-based perfect masking checking and Column 7 shows the time for SMT-based perfect masking checking.

Results in Table 4 indicate that most of the time is spent on computing *don't care* variables and SemdR, while the time taken by the SMT solver to conduct model-counting (SAT#) is relatively small. In contrast, the original SMT-based approach spent a large amount of time on the static analysis part, which performs code partitioning and applies the heuristic rules (cf. Section 4.1). This explains why our new method is more efficient than the original SMT-based approach. Moreover, the time of our new method can be improved further by disabling the SemdR computing on P12-P17.

Table 3. Statistics: Number of Nodes in Different Distribution Types, Where Column ♯T for Each $T \in \{$RUD, SID, CST, NPM$\}$ Denotes the Number of T-typed Nodes, Column ♯*Resolved* UKD *By SMT* (Respectively, ♯*Resolved* UKD *By* SemdR) Denotes the Number of UKD-typed Nodes That Are Resolved by the SMT-based Approach (Respectively, Semantic Type inference), and Column ♯UKD *to* NPM (Respectively, ♯UKD *to* SID) Denotes the Number of UKD-typed Nodes That Are Refined to NPM Type (Respectively, SID Type)

Name	♯RUD	♯SID	♯CST	♯NPM	♯Resolved UKD by SMT	♯Resolved UKD by SemdR	♯UKD to NPM	♯UKD to SID
P1	16	0	0	16	16	0	16	0
P2	16	0	0	8	8	0	8	0
P3	6	0	0	0	0	0	0	0
P4	6	0	0	0	0	1	0	0
P5	6	2	0	0	2	1	0	2
P6	4	3	0	2	2	0	2	0
P7	5	5	0	1	1	1	1	0
P8	11	4	0	3	3	0	3	0
P9	12	4	0	2	2	0	2	0
P10	20	6	1	1	2	1	1	1
P11	19	7	1	1	3	1	1	2
P12	190400	6400	0	0	0	0	0	0
P13	185600	6400	0	4800	4800	0	4800	0
P14	187200	6400	0	3200	3200	0	3200	0
P15	188800	8000	0	1600	3200	0	1600	1600
P16	185600	6400	0	4800	4800	0	4800	0
P17	185600	1600	0	17600	12800	0	12800	0

6.3 Experimental Results on Checking a QMS Requirement

In the previous subsection, we have shown results of verifying perfect masking, which can be seen as a special case of checking the QMS requirement of 1. In this subsection, we report the results of two more experiments on checking the more general QMS requirements to understand how the QMS requirements affect performance.

In the first experiment, we check all benchmark programs against the QMS requirement 0.5. Table 5 shows the result. Column 2 and Column 3 show the number of nodes that satisfy and dissatisfy the QMS requirement, respectively. Columns 4 and 5 show the results of the SMT-based approach from Reference [41], including the number of nodes that are checked by the SMT-based approach and the corresponding time. Similarly, Columns 6–8 show the results of QMSInfer including the number of nodes that are checked by calling the SMT solver, the number of nodes that skipped due to perfect masking proved by calling the SMT solver, and the corresponding time.

Columns 2 and 3 show that six programs in our benchmarks do not satisfy the QMS requirement 0.5, which is significantly different from the results of checking the QMS requirement 1 (cf. Table 3). Compared with the SMT-based approach [41], CheckQMS takes less time on the large programs (i.e. P12–P17), which confirms that our refinement-based approach significantly improves the efficiency. Columns 4, 6, and 7 show that perfectly masked nodes can be skipped for checking the QMS requirements.

In the second experiment, we check the larger programs, P14, P15, and P16, against the QMS requirement values ranged from 0.1 to 1.0 with step 0.1. Figure 8 shows the number of unsat nodes

Table 4. Statistics: Where Column ♯*SMT Calls* and *time* in Column *Computing* SemdR (Respectively, Columns *Computing don't care* and *Checking Perfect Masking*) Denote the Number of Calls to SMT Solver and Corresponding Execution Time During the Computation of SemdR (don't Care Random Variables and Checking Perfect Masking by the SMT-based Approach)

Name	Computing SemdR		Computing don't care		Checking perfect masking	
	♯SMT Calls	Time	♯SMT Calls	Time	♯SMT Calls	Time
P1	0	0s	0	0s	16	0.41s
P2	16	0.38s	8	0.15s	8	0.20s
P3	0	0s	0	0s	0	0s
P4	3	0s	1	0s	0	0s
P5	6	0.11s	5	0.10s	2	0.06s
P6	8	0.13s	0	0.10s	2	0.04s
P7	8	0.06s	1	0.03s	1	0.02s
P8	12	0.17s	2	0.10s	3	0.04s
P9	12	0.17s	0	0.06s	2	0.04s
P10	11	0.14s	1	0.10s	2	0.04s
P11	15	0.16s	2	0.15s	3	0.06s
P12	0	0s	0	0s	0	0s
P13	105600	29m 42s	52800	16m 3s	4800	1m 25s
P14	19200	27m 51s	148416	26m 13s	3200	1m 15s
P15	17600	21m 7s	148288	35m 35s	3200	1m 47s
P16	16000	27m 40s	174016	34m 3s	4800	1m 37s
P17	6403	19m 8s	317760	85m 57s	12800	4m 4s

(i.e., nodes that do not satisfy the QMS requirement) and the corresponding time. In the first part of Figure 8, the x-axis is the QMS requirement, and the y-axis is the number of unsat nodes. The result shows that the three programs have the same numbers of unsat nodes when the QMS value is less than or equal to 0.5 or greater than 0.6. Furthermore, there is a significantly increase from 0.5 to 0.6. In the second part of Figure 8, the x-axis is the QMS requirement, and the y-axis is the time. The result shows that there is no explicit corelation between the time and the number of unsat nodes.

6.4 Experimental Results on Computing the QMS Value

In this subsection, we conduct two experiments. The first experiment computes the QMS values for all intermediate computation results of each program, and the second experiment computes only the minimal QMS value.

Table 6 shows the experimental results of computing the QMS values. Columns 2–6 (respectively, Columns 7–11) show the statistics of the SMT-based approach [41] (respectively, our QMSInfer approach), including the total number of iterations during the computation of QMS values, the total time, as well as the minimal, maximal, and average of the QMS values. Columns 12–14 show the difference of the minimal, maximal, and average QMS values between two approaches, respectively.

Compared with the SMT-based approach [41], our approach QMSInfer takes significant fewer iterations and less time, especially on the larger programs (i.e., P12–P17), as our binary search depends on the number of semantically dependent random variables and thus can be more

Table 5. Experimental Results of Checking the Benchmark Programs Against the QMS Requirement ≥ 0.5, Where the Column ♯SAT (Respectively, ♯UNSAT) Denotes the Number of Nodes That Satisfies (Respectively, Does Not Satisfy) the QMS Requirement 0.5, Column ♯*QMS By SMT* Denotes the Number of Nodes That Are Checked by the SMT-based Approach, and Column ♯*P.M. By SMT* Denotes the Number of Nodes That Skipped Due to Perfect Masking Proved by Calling the SMT Solver

Name	♯Nodes		SMT-based [41]		CheckQMS		
	♯SAT	♯UNSAT	♯QMS By SMT	Time	♯QMS By SMT	♯P.M. By SMT	Time
P1	16	16	16	0.69s	16	0	0.66s
P2	16	8	8	0.39s	8	0	0.64s
P3	6	0	0	0s	0	0	0s
P4	6	0	3	0.20s	0	0	≈0s
P5	8	0	3	0.21s	0	2	0.20s
P6	9	0	2	0.13s	2	0	0.20s
P7	11	0	2	0.15s	1	0	0.19s
P8	17	1	3	0.21s	3	0	0.33s
P9	18	0	2	0.15s	2	0	0.28s
P10	28	0	2	0.19s	1	1	0.26s
P11	28	0	3	0.28s	1	2	0.37s
P12	196800	0	0	91m 42s	0	0	4.34s
P13	192000	4800	4800	97m 2s	4800	0	47m 57s
P14	196800	0	3200	116m 59s	3200	0	53m 23s
P15	198400	0	3200	129m 24s	1600	1600	55m 33s
P16	195200	1600	4800	127m 44s	4800	0	60m 59s
P17	192000	12800	16000	351m 50s	12800	0	109m 4s

efficient than the step of 0.01 used in the SMT-based approach [41]. Columns 12–14 indicate that the difference on the minimal QMS and average QMS between two approaches are not larger than 0.01, which explains why the two approaches obtain almost the same results. Since the number of semantically dependent random variables for each intermediate computation result is usually small, the step 0.01 in the SMT-based approach [41] is accurate enough to approach the real QMS value.

Table 7 shows the experimental results of computing the minimal QMS value. Columns 2–4 (respectively, Columns 5–7) show statistics of MinQMSInfer approach (respectively, QMSInfer approach), including the total number of iterations to obtain the minimal QMS value, the time for computing the minimal QMS value (excluding perfect masking checking), and the total time. Compared with QMSInfer, MinQMSInfer takes significant fewer iterations and less time to obtain the minimal QMS value. Column 8 shows the minimal QMS value computed.

7 RELATED WORK

In this section, we review related work on masking countermeasures in general, as well as existing techniques on the verification of perfect masking, quantitative estimation of information leakage, and the detection/mitigation of other types of side-channel leaks.

7.1 Masking

Many masking countermeasures [22, 26, 49, 53, 59, 61, 64, 72, 74–76] have been published in the past two decades: Although they differ in adversary models, masking schemes, cryptographic algorithms, and compactness, these countermeasures are often manually designed for specific

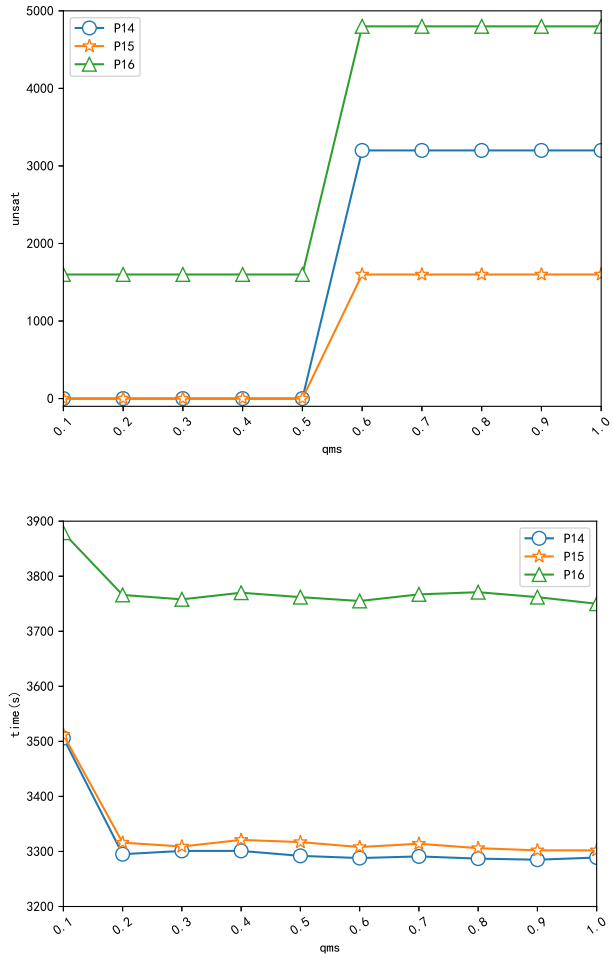

Fig. 8. The number of unsatisfiable nodes (above figure) and the time (below figure) with respect to the QMS requirements.

cryptographic algorithms. In this context, the common problem is the lack of efficient and automated tools for proving their correctness [34, 35]. Our work aims to bridge the gap.

Another line of existing work is mitigating side-channel attacks automatically [1, 9, 14, 23, 38, 62, 82]. For example, techniques proposed in References [1, 9, 14, 62] rely on compiler-like pattern matching, whereas the ones proposed in References [23, 38, 82] use inductive program synthesis. Both types of techniques, however, are orthogonal to our work reported in this article. Thus, it would be interesting to investigate whether our new method can aid in the synthesis of better masking countermeasures, as done in Reference [38].

7.2 Perfect Masking Verification

There are two types of existing methods for verifying perfect masking. One type is simulation-based methods [4, 48, 78], which are able to detect side-channel leaks but not prove their absence. The other type is formal verification methods [8, 9, 15, 17, 20, 21, 33, 39, 40, 45, 65], which are able to prove the absence of side-channel leaks. However, as we have explained earlier, these existing

Table 6. Experimental Results of Computing the QMS Values, Where Column ♯*Iter* Gives the Total Number of Iterations during Binary Search, Columns *Min*, *Max*, and *Arg* Give the Minimal, Maximal, and Average QMS Values, and Columns *Min Diff*, *Max Diff*, and *Arg Diff* Give the Difference of Minimal, Maximal, and Average QMS Values between the Prior SMT-based Method [41] and Our QMSInfer

Name	SMT-based [41]					QMSInfer					Min	Max	Avg
	♯Iter	Time	Min	Max	Arg	♯Iter	Time	Min	Max	Arg	Diff	Diff	Diff
P1	1600	27.40s	0.00	1.00	0.50	0	0.36s	0.00	1.00	0.50	0.0	0.0	0.0
P2	800	13.94s	0.00	1.00	0.67	0	0.60s	0.00	1.00	0.67	0.0	0.0	0.0
P3	0	0s	1.00	1.00	1.00	0	0s	1.00	1.00	1.00	0.0	0.0	0.0
P4	0	0.15s	1.00	1.00	1.00	0	0.06s	1.00	1.00	1.00	0.0	0.0	0.0
P5	0	0.15s	1.00	1.00	1.00	0	0.21s	1.00	1.00	1.00	0.0	0.0	0.0
P6	100	1.90s	0.51	1.00	0.89	6	0.27s	0.50	1.00	0.89	0.01	0.0	0.0
P7	50	0.98s	0.51	1.00	0.96	2	0.20s	0.50	1.00	0.95	0.01	0.0	0.01
P8	200	3.69s	0.00	1.00	0.89	6	0.39s	0.00	1.00	0.89	0.0	0.0	0.0
P9	100	1.92s	0.51	1.00	0.95	6	0.34s	0.50	1.00	0.94	0.01	0.0	0.01
P10	50	1.07s	0.51	1.00	0.98	3	0.30s	0.50	1.00	0.98	0.01	0.0	0.0
P11	50	1.23s	0.51	1.00	0.98	3	0.40s	0.50	1.00	0.98	0.01	0.0	0.0
P12	0	93m 4s	1.00	1.00	1.00	0	4.50s	1.00	1.00	1.00	0.0	0.0	0.0
P13	480000	239m 44s	0.00	1.00	0.98	0	45m 55s	0.00	1.00	0.98	0.0	0.0	0.0
P14	160000	181m 27s	0.51	1.00	0.99	9600	55m 18s	0.50	1.00	0.99	0.01	0.0	0.0
P15	80000	170m 30s	0.51	1.00	1.00	4800	56m 46s	0.50	1.00	1.00	0.01	0.0	0.0
P16	320000	232m 33s	0.00	1.00	0.98	6400	61m 32s	0.00	1.00	0.98	0.0	0.0	0.0
P17	1440000	1057m 1s	0.00	1.00	0.93	4800	111m 4s	0.00	1.00	0.94	0.0	0.0	0.01

formal verification methods are either fast but inaccurate (e.g., type-inference-based techniques) or accurate but slow (e.g., model-counting-based techniques).

More specifically, Bayrak et al. [15] developed a leak detector, which checks if a computation result is *logically* dependent of the secret data and, at the same time, *logically* independent of any random variable used for masking the secret data. Their method is fast, but not accurate, in that many leaky nodes could be incorrectly classified as leakage free [39, 40].

Barthe et al. leveraged the notion of t-noninterference [8] from probabilistic programs to verify perfect masking, and they proposed a syntactic type-inference method to prove t-noninterference by exploiting the unique characteristics of invertible operations such as $\oplus$. Purely algebraic laws were used to normalize expressions of intermediate computation results, so that the rules of invertible functions can be applied. However, since expression normalization is costly, it is applied only once for each intermediate computation result.

The notion of t-noninterference was extended later [9] for compositional verification of perfect masking. That is, it allows the authors to prove the security of smaller code sequences (called gadgets) when composed with other code parts (gadgets satisfying a stronger version of t-noninterference can be freely composed with other gadgets without interfering). More recently, (strong) t-noninterference was also extended with glitches [10]. However, the problem is that not all masking algorithms are composable and thus can be verified using this technique. Following [8], Bisi et al. [20] proposed a technique for verifying higher-order masking, but the technique was limited to linear operations only. Ouahma et al. also generalized the approach of Reference [8] to verify assembly-level code [21].

Coron proposed two alternative approaches to prove (strong) t-noninterference [33]. The first one is similar to Reference [8] but uses the Common Lisp language. The second one uses

Table 7. Experimental Results of Computing the Minimal QMS Value, Where Column ♯*Iter* Gives the Total Number of Iterations during Binary Search, Columns *Time4QMS* Give the Time Used for Computing QMS Values (Excluding the Verification of Perfect Masking), and Column *Min QMS Value* Gives the Minimal QMS Value Computed

Name	MinQMSInfer			QMSInfer			Min QMS
	♯Iter	Time4QMS	Total time	♯Iter	Time4QMS	Total time	Value
P1	0	0s	0.02s	0	0s	0.36s	0.00
P2	0	0s	0.09s	0	0s	0.60s	0.00
P3	0	0s	0s	0	0s	0s	1.00
P4	0	0s	0.25s	0	0s	0.06s	1.00
P5	0	0s	0.28s	0	0s	0.21s	1.00
P6	4	0.08s	0.24s	6	0.10s	0.27s	0.50
P7	2	0.03s	0.20s	2	0.03s	0.20s	0.50
P8	4	0.07s	0.35s	6	0.11s	0.39s	0.00
P9	4	0.07s	0.31s	6	0.11s	0.34s	0.50
P10	3	0.06s	0.29s	3	0.06s	0.30s	0.50
P11	3	0.07s	0.37s	3	0.06s	0.40s	0.50
P12	0	0s	4.11s	0	0s	4.50s	1.00
P13	0	0s	0.76s	0	0s	45m 55s	0.00
P14	3202	1m 24s	53m 37s	9600	3m 49s	55m 18s	0.50
P15	1602	59.43s	54m 57s	4800	2m 33s	56m 46s	0.50
P16	3	0.05s	2.54s	6400	2m 4s	61m 32s	0.00
P17	0	0s	0.66s	4800	1m 14s	111m 4s	0.00

elementary transformations to make the targeted program verifiable using (strong) t-noninterference. Faust et al. also generalized the notion of (strong) t-noninterference with glitches [45] for hardware, but, to our knowledge, no implementation or evaluation exists.

Bhasin et al. [17] proposed a Fourier transform-based approach to estimate the side-channel attack resistance of circuits. Their approach uses a SAT solver to construct low-weight functions of a certain resistance order, but has not been used to evaluate existing implementations of cryptographic functions. A similar idea was proposed by Bloem at al. [21], which takes glitches into account and proves perfect masking by estimating the non-zero Fourier coefficients of the functions in hardware.

It is worth noting that all the above formal verification methods are incomplete in that it is possible for programs to be secure and, at the same time, cannot be verified by these methods. In contrast, the model-counting-based method proposed by Eldib et al. [39, 40] is both sound and complete, but also significantly less scalable, because the size of the first-order logic formulas that they need to construct and solve are exponential in the number of random variables used for masking the secret data.

Our gradual refinement of type-inference rules was inspired by recent works on proving probabilistic non-interference [8, 9, 21, 33, 65]. However, our method differs from them in that their type-inference rules are always syntactic and fixed, whereas our type-inference rules are both semantic and can be gradually refined using SMT solver-based satisfiability-checking and model-counting (SAT and SAT#).

An alternative way of solving the model-counting problem [5, 27, 28, 47] is to use satisfiability modulo counting, which is a generalization of the satisfiability problem of SMT with counting constraints [46]. Toward this end, Fredrikson and Jha [46] have developed an efficient decision

procedure for linear integer arithmetic (LIA) based on Barvinok's algorithm [13] and also applied their approach to differential privacy. However, more research is needed to apply similar techniques to cryptographic algorithms, where non-linear functions are widely used.

7.3 Quantitative Estimation of Information Leakage

The notion of QMS was proposed by Eldib et al. in References [41, 42] for quantifying the resistance against power side-channel attacks. As mentioned earlier, their algorithm computes an approximation of the QMS by directly searching for a value between 0 to 1 with step 0.01, whereas our new method computes the more accurate QMS value using a binary search that takes into account the number of random variables in the intermediate computation results.

There exist other security metrics for quantitative information flow analysis such as Shannon-entropy, mutual information, and min-entropy from information theory [19, 31, 57, 70, 71, 77, 81]. The quantitative information flow framework has also been specialized to perform side-channel analysis [58, 66, 68, 79]. In general, these metrics are used to quantify how much information is being leaked and the success rate or guessing entropy, while QMS is used to quantify how many power measurement traces are required to successfully break the countermeasure. Moreover, in the context of QMS, program inputs are partitioned into public and private variables, which means the leakage should be understood as conditional mutual information as mentioned in, e.g., Reference [58].

7.4 Other Types of Side Channels

In addition to detecting and mitigating power side-channel attacks, there are techniques for mitigating other types of side-channel attacks, where the side channels can be in the form of the CPU time [2, 3, 7, 25, 30, 54, 67, 69, 84], faults [11, 18, 24, 43] and cache behaviors [12, 29, 32, 37, 50, 51, 56, 80, 83, 85]. Since each type of side-channel has unique characteristics, in general, these existing techniques are orthogonal to our work. Nevertheless, it remains an open problem whether our refinement-based techniques, in principle, can be used to improve the accuracy and scalability of verification in these contexts.

8 CONCLUSIONS AND FUTURE WORK

We have presented a refinement-based approach for verifying and quantifying of side-channel resistance of masked software implementations. Our method relies on a set of semantic inference rules to reason about distribution types of intermediate computation results, coupled with an SMT solver-based procedure for gradually refining these types to increase accuracy. We have implemented our approach in a tool and demonstrated its efficiency and effectiveness on cryptographic benchmarks. Our results show that it outperforms state-of-the-art techniques.

For future work, we plan to evaluate our type inference systems for higher-order masking, extend it to handle integer programs as opposed to bit-blasting them to Boolean programs, and investigate the synthesis of masking countermeasures based on our verification method.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the ACM/IEEE Design Automation Conference*. 77–82.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*. 53–70.

[3] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. 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*. 362–375.

[4] Victor Arribas, Svetla Nikova, and Vincent Rijmen. 2017. VerMI: Verification tool for masked implementations. In *Proceedings of the IACR Cryptology ePrint Archive* (2017), 1227.

[5] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *Proceedings of the International Conference on Computer Aided Verification*. 255–272.

[6] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2014. On the cost of lazy engineering for masked software implementations. In *Proceedings of the International Conference on Smart Card Research and Advanced Applications (CARDIS'14)*. 64–81.

[7] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 193–204.

[8] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. 2015. Verified proofs of higher-order masking. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic (EUROCRYPT'15)*. 457–485.

[9] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *Proceedings of the ACM Conference on Computer and Communications Security*. 116–129.

[10] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. 2018. maskVerif: A formal tool for analyzing software and hardware masked implementations. *IACR Cryptology ePrint Archive* 2018 (2018), 562.

[11] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. 2014. Synthesis of fault attacks on cryptographic implementations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 1016–1027.

[12] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. 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*. 140–158.

[13] Alexander I. Barvinok. 1994. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779.

[14] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the ACM/IEEE Design Automation Conference*. 230–235.

[15] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated verification of software power analysis countermeasures. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*. 293–310.

[16] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. 2013. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf.

[17] Shivam Bhasin, Claude Carlet, and Sylvain Guilley. 2013. Theory of masking with codewords in hardware: Low-weight dth-order correlation-immune Boolean functions. *IACR Cryptology ePrint Archive* 2013 (2013), 303.

[18] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO'97)*. 513–525.

[19] Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. 2018. Scalable approximation of quantitative information flow in programs. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'18)*. 71–93.

[20] Elia Bisi, Filippo Melzani, and Vittorio Zaccaria. 2017. Symbolic analysis of higher-order side channel countermeasures. *IEEE Trans. Comput.* 66, 6 (2017), 1099–1105.

[21] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. 2018. 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*. 321–353.

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

[23] Arthur Blot, Masaki Yamamoto, and Tachio Terauchi. 2017. Compositional synthesis of leakage resilient programs. In *Proceedings of the International Conference on Principles of Security and Trust*. 277–297.

[24] Jakub Breier, Xiaolu Hou, and Yang Liu. 2017. Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code. Cryptology ePrint Archive, Report 2017/829.

[25] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the International Conference on Software Engineering*. 424–425.

[26] D. Canright and Lejla Batina. 2008. A very compact "perfectly masked" S-Box for AES. In *Proceedings of the International Conference on Applied Cryptography and Network Security*. 446–459.

[27] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-aware sampling and weighted model counting for SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1722–1730.

[28] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A scalable approximate model counter. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 200–216.

[29] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the information leak in cache attacks via symbolic execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 25–35.

[30] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 875–890.

[31] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2014. LeakWatch: Estimating information leakage from java programs. In *Proceedings of the 19th European Symposium on Research in Computer Security*. 219–236.

[32] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Precise cache timing analysis via symbolic execution. In *Proceedings of the IEEE Symposium on Real-Time and Embedded Technology and Applications*. 293–304.

[33] Jean-Sébastien Coron. 2018. Formal verification of side-channel countermeasures via elementary circuit transformations. In *Proceedings of the 16th International Conference on Applied Cryptography and Network Security*. 65–82.

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

[35] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. 2013. Higher-order side channel security and mask refreshing. In *Proceedings of the International Workshop on Fast Software Encryption (FSE'13)*. 410–424.

[36] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 337–340.

[37] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the USENIX Security Symposium*. 431–446.

[38] Hassan Eldib and Chao Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *Proceedings of the International Conference on Computer Aided Verification*. 114–130.

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

[40] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. 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*. 62–77.

[41] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014. QMS: Evaluating the side-channel resistance of masked software from source code. In *Proceedings of the ACM/IEEE Design Automation Conference*. 209:1–6.

[42] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. 2015. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. CAD Integr. Circ. Syst.* 34, 10 (2015), 1558–1568.

[43] Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of fault-attack countermeasures for cryptographic circuits. In *Proceedings of the International Conference on Computer Aided Verification*. 343–363.

[44] Christophe Clavier et al. 2014. Practical improvements of side-channel attacks on AES: Feedback from the 2nd DPA contest. *J. Cryptogr. Eng.* 4, 4 (2014), 259–274.

[45] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. 2017. Composable masking schemes in the presence of physical defaults and the robust probing model. *IACR Cryptology ePrint Archive* 2017 (2017), 711.

[46] Matthew Fredrikson and Somesh Jha. 2014. Satisfiability modulo counting: A new approach for analyzing privacy properties. In *Proceedings of the ACM/IEEE Symposium on Logic in Computer Science*. 42:1–42:10.

[47] Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. 2017. Maximum model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 3885–3892.

[48] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. 2011. A testing methodology for side channel resistance validation. In *Proceedings of the NIST Non-invasive Attack Testing Workshop*.

[49] Louis Goubin. 2001. A sound method for switching between boolean and arithmetic masking. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*. 3–15.

[50] Philipp Grabher, Johann Großschädl, and Dan Page. 2007. Cryptographic side-channels from low-power cache memory. In *Proceedings of the IMA International Conference on Cryptography and Coding*. 170–184.

[51] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[52] Gary D. Hachtel and Fabio Somenzi. 1996. *Logic Synthesis and Verification Algorithms*. Kluwer.

[53] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private circuits: Securing hardware against probing attacks. In *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO'03)*. 463–481.

[54] Paul C. Kocher. 1996. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO'96)*. 104–113.

[55] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO'99)*. 388–397.

[56] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *Proceedings of the International Conference on Computer Aided Verification*. 564–580.

[57] Pasquale Malacaria and Jonathan Heusser. 2010. 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'10)*. 87–134. DOI: https://doi.org/10.1007/978-3-642-13678-8_3

[58] Pasquale Malacaria, M. H. R. Khouzani, Corina S. Pasareanu, Quoc-Sang Phan, and Kasper Søe Luckow. 2018. Symbolic side-channel analysis for probabilistic programs. *IACR Cryptology ePrint Archive* 2018 (2018), 329.

[59] Thomas S. Messerges. 2000. Securing the AES finalists against power analysis attacks. In *Proceedings of the International Workshop on Fast Software Encryption*. 150–164.

[60] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from xilinx Virtex-II FPGAs. In *Proceedings of the ACM Conference on Computer and Communications Security*. 111–124.

[61] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. 2011. 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'11)*. 69–88.

[62] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*. 58–75.

[63] Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. 2012. RSM: A small and fast countermeasure for AES, secure against 1st and 2nd-order zero-offset SCAs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'12)*. 1173–1178.

[64] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. 2005. A side-channel analysis resistant description of the AES S-Box. In *Proceedings of the International Workshop on Fast Software Encryption*. 413–423.

[65] Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz. 2017. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *Security Proofs for Embedded Systems*.

[66] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using symbolic execution and max-SMT. In *Proceedings of the IEEE 29th Computer Security Foundations Symposium (CSF'16)*. 387–400.

[67] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using symbolic execution and max-SMT. In *Proceedings of the IEEE Computer Security Foundations Symposium*. 387–400.

[68] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of adaptive side-channel attacks. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF'17)*. 328–342. DOI: https://doi.org/10.1109/CSF.2017.8

[69] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of adaptive side-channel attacks. In *Proceedings of the IEEE Computer Security Foundations Symposium*. 328–342.

[70] Quoc-Sang Phan and Pasquale Malacaria. 2014. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'14)*. 283–292. DOI: https://doi.org/10.1145/2590296.2590328

[71] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo d'Amorim. 2014. Quantifying information leaks using reliability analysis. In *Proceedings of the 2014 International Symposium on Model Checking of Software (SPIN'14)*. 105–108. DOI: https://doi.org/10.1145/2632362.2632367

[72] Emmanuel Prouff and Matthieu Rivain. 2013. Masking against side-channel attacks: A formal security proof. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*. 142–159.

[73] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Proceedings of the International Conference on Research in Smart Cards (E-smart'01)*. 200–210.

[74] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating masking schemes. In *Proceedings of the Annual Cryptology Conference on Advances in Cryptology (CRYPTO'15)*. 764–783.

[75] Matthieu Rivain and Emmanuel Prouff. 2010. Provably secure higher-order masking of AES. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*. 413–427.

[76] Kai Schramm and Christof Paar. 2006. Higher order masking of the AES. In *Proceedings of the RSA Conference on Topics in Cryptology (CT-RSA'06)*. 208–225.

[77] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures, Held as Part of the Joint European Conferences on Theory and Practice of Software*. 288–302.

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

[79] François-Xavier Standaert, Tal Malkin, and Moti Yung. 2009. A unified framework for the analysis of side-channel key recovery attacks. In *Proceedings of 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'09)*. 443–461.

[80] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: A cache timing analysis framework via LLVM transformation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.

[81] Celina G. Val, Michael A. Enescu, Sam Bayless, William Aiello, and Alan J. Hu. 2016. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'16)*. 31–46.

[82] Chao Wang and Patrick Schaumont. 2017. Security by compilation: An automated approach to comprehensive side-channel resistance. *SIGLOG News* 4, 2 (2017), 76–89.

[83] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying cache-based timing channels in production software. In *Proceedings of the USENIX Security Symposium*. 235–252.

[84] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 15–26.

[85] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[86] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. 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*. 157–177.

Received August 2018; revised January 2019; accepted April 2019