SCInfer: Refinement-based Verification of Software Countermeasures against Side-Channel Attacks ⋆

Jun Zhang[1], Pengfei Gao[1], Fu Song[1]✉, and Chao Wang[2]

[1] ShanghaiTech University, Shanghai, China
[2] University of Southern California, Los Angeles, CA, USA

Abstract. Power side-channel attacks, capable of deducing secret using statistical analysis techniques, have become a serious threat to devices in cyber-physical systems and the Internet of things. Random masking is a widely used countermeasure for removing the statistical dependence between secret data and side-channel leaks. Although there are techniques for verifying whether software code has been 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 syntactic type inference based approaches and more scalable than prior model-counting based approaches using SAT or SMT solvers. Indeed, it can be viewed as a gradual refinement of a set of semantic 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 have implemented our method in a tool and evaluated it on cryptographic benchmarks including AES and MAC-Keccak. The results show that our method significantly outperforms state-of-the-art techniques in terms of both accuracy and scalability.

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 has access to the input and output, but not internals, of the implementation. Unfortunately, in practice, attackers may recover cryptographic keys 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 [38], power consumption [39] and electromagnetic radiation [49]. Among them, *differential power analysis* (DPA) is an extremely popular and effective class of attacks [42,30].

To thwart DPA attacks, *masking* has been proposed to break the statistical dependence between secret data and side-channel leaks through randomization. Although various masked implementations have been proposed, e.g., for AES or its non-linear components (S-boxes) [15,52,51,37], checking if they are correct is always tedious and error-prone. Indeed, there are published implementations [51,52] later shown to be incorrect [21,22]. Therefore, formally verifying these countermeasures is important.

⋆ This work was supported primarily by the National Natural Science Foundation of China (NSFC) grants 61532019 and 61761136011. Chao Wang was supported by the U.S. National Science Foundation (NSF) grant CNS-1617203.

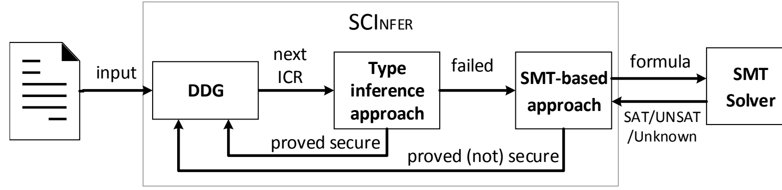

Fig. 1. Overview of SCInfer, where "ICR" denotes the intermediate computation result.

Previously, there are two types of verification methods for masking countermeasures [54]: one is type inference based [44,10] and the other is model counting based [27,26]. Type inference based methods [44,10] are fast and sound, meaning they can quickly prove the computation is leakage free, e.g., if the result is syntactically independent of the secret data or has been masked by random variables not used elsewhere. However, syntactic type inference is *not* complete in that it may report *false positives*. In contrast, model counting based methods [27,26] are sound and complete: they check if the computation is statistically independent of the secret [15]. However, due to the inherent complexity of model counting, they can be extremely slow in practice.

The aforementioned gap, in terms of both accuracy and scalability, has not been bridged by more recent approaches [6,47,13]. For example, Barthe et al. [6] proposed some inference rules to prove masking countermeasures based on the observation that certain operators (e.g., XOR) are *invertible*: in the absence of such operators, purely algebraic laws can be used to normalize expressions of computation results to apply the rules of invertible functions. This normalization is applied to each expression once, as it is costly. Ouahma et al. [47] introduced a linear-time algorithm based on finer-grained syntactical inference rules. A similar idea was explored by Bisi et al. [13] for analyzing higher-order masking: like in [6,47], however, the method is not complete, and does not consider non-linear operators which are common in cryptographic software.

Our contribution. We propose a refinement based approach, named SCInfer, to bridge the gap between prior techniques which are either fast but inaccurate or accurate but slow. Fig. 1 depicts the overall flow, where the input consists of the program and a set of variables marked as *public*, *private*, or *random*. 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. Next, we check if all intermediate computation results are perfectly masked according to their types. If any of them cannot be resolved in this way, we invoke an SMT solver based refinement procedure, which leverages either satisfiability (SAT) solving or model counting (SAT#) to prove leakage freedom. In both cases, the result is fed back to improve the type system. Finally, based on the refined type inference rules, we continue to analyze other intermediate computation results.

Thus, SCInfer can be viewed as a synergistic integration of a semantic rule based approach for inferring *distribution types* and an SMT solver based approach for refining these inference rules. Our type inference rules (Section 3) are inspired by Barthe et al. [6] and Ouahma et al. [47] in that they are designed to infer distribution types of intermediate computation results. However, there is a crucial difference: their inference

rules are syntactic with fixed accuracy, i.e., relying solely on structural 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 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 times, 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 propose to leverage SMT solvers 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 if the computation is perfectly masked. Finally, by feeding solver results back to the type inference system, we can gradually improve its accuracy. Thus, overall, the method is both sound and complete.

We have implemented our method in a software tool named SCInfer and evaluated it on publicly available benchmarks [27,26], which implement various cryptographic algorithms such as AES and MAC-Keccak. Our experiments show SCInfer 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 (SAT) based analysis needs to be applied to few left-over cases. Only in rare cases, the most heavyweight 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 method for gradually refining the type inference system using SMT solver based analysis, to ensure the overall method is complete.
- We implement the proposed techniques in a tool named SCInfer and demonstrate its efficiency and effectiveness on cryptographic benchmarks.

The remainder of this paper is organized as follows. After reviewing the basics in Section 2, we present our semantic type inference system in Section 3 and our refinement method in Section 4. Then, we present our experimental results in Section 5 and comparison with related work in Section 6. We give our conclusions in Section 7.

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 [15,27,26], we assume that the program P implements a cryptographic function, e.g., $\boldsymbol{c} \leftarrow P(\boldsymbol{p}, \boldsymbol{k})$ where $\boldsymbol{p}$ is the plaintext, $\boldsymbol{k}$ is the secret key and $\boldsymbol{c}$ is the ciphertext. Inside P, random variable $\boldsymbol{r}$ may be used to mask the secret key while maintaining the input-output behavior of P. Therefore, P may be viewed as a probabilistic program. Since loops, function calls, and branches may be removed via automated rewriting [27,26] and integer variables may be converted to bits, 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 $\boldsymbol{k}$ (resp. $\boldsymbol{r}$) be the set of secret (resp. random) bits, $\boldsymbol{p}$ the public bits, and $\boldsymbol{c}$ the variables storing intermediate results. Thus, the set of variables is $V = \boldsymbol{k} \cup \boldsymbol{r} \cup \boldsymbol{p} \cup \boldsymbol{c}$. In addition, the program uses a set op of operators including negation ($\neg$), and ($\wedge$), or ($\vee$), and exclusive-or ($\oplus$). A *computation* of P is a sequence $c_1 \leftarrow \mathbf{i}_1(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r}); \cdots ; c_n \leftarrow \mathbf{i}_n(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ where, for each $1 \leq i \leq n$, the value of $\mathbf{i}_i$ is expressed in terms of $\boldsymbol{p}$, $\boldsymbol{k}$ and $\boldsymbol{r}$. Each random bit in $\boldsymbol{r}$ is uniformly distributed in $\{0, 1\}$; the sole purpose of using them in P is to ensure that $c_1, \cdots c_n$ are statistically independent of the secret $\boldsymbol{k}$.

```
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 }
```

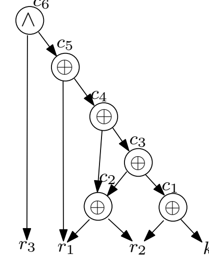

Fig. 2. An example for masking countermeasure.

Data dependency graph (DDG). Our internal representation of 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.

- $N = L \uplus L_V$, where L is the set of instructions in P and L_V is the set of terminal nodes: $l_v \in L_V$ corresponds to a variable or constant $v \in \boldsymbol{k} \cup \boldsymbol{r} \cup \boldsymbol{p} \cup \{0, 1\}$.
- $E \subseteq N \times N$ contains edge (l, l') if and only if $l : c = x \circ y$, where either x or y is defined by l'; or $l : c = \neg x$, where x is defined by l';
- 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 $\lambda(l) = (v, \perp)$ for each terminal node l_v.

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.lft to denote the left child of l, and l.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 $c = \lambda_1(l)$. Let $|P|$ denote the total number of nodes in the DDG.

Fig. 2 shows an example where $\boldsymbol{k} = \{k\}$, $\boldsymbol{r} = \{r_1, r_2, r_3\}$, $\boldsymbol{c} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ and $\boldsymbol{p} = \emptyset$. On the left is a program written in a C-like language except that $\oplus$ denotes XOR and $\wedge$ denotes AND. On the right is the DDG, where

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

Let supp : $N \rightarrow \boldsymbol{k} \cup \boldsymbol{r} \cup \boldsymbol{p}$ be a function mapping each node l to its support variables. That is, supp(l) $= \emptyset$ if $\lambda_1(l) \in \{0, 1\}$; supp($l$) $= \{x\}$ if $\lambda_1(l) = x \in \boldsymbol{k} \cup \boldsymbol{r} \cup \boldsymbol{p}$; and supp($l$) $=$ supp(l.lft) $\cup$ supp(l.rgt) otherwise. Thus, the function returns a set of variables that $\lambda_1(l)$ depends upon structurally.

Given a node l whose corresponding expression e is defined in terms of variables in V, we say that e is semantically dependent on a variable $r \in V$ 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 $x \in V \setminus \{r\}$, and the values of e differ under π_1 and π_2.

Let $\mathtt{semd} : N \rightarrow \boldsymbol{r}$ be a function such that *semd*(l) denotes the set of *random variables* upon which the expression e of l semantically depends. Thus, $\mathtt{semd}(l) \subseteq \mathtt{supp}(l)$; and for each $r \in \mathtt{supp}(l) \setminus \mathtt{semd}(l)$, we know $\lambda_1(l)$ is semantically independent of r. More importantly, there is often a gap between $\mathtt{supp}(l) \cap \boldsymbol{r}$ and $\mathtt{semd}(l)$, namely $\mathtt{semd}(l) \subsetneq \mathtt{supp}(l) \cap \boldsymbol{r}$, which is why our gradual refinement of semantic type inference rules can outperform methods based solely on syntactic type inference.

Consider the node l_{c_4} in Fig. 2: we have $\mathtt{supp}(l_{c_4}) = \{r_1, r_2, k\}$, $\mathtt{semd}(l_{c_4}) = \{r_2\}$, and $\mathtt{supp}(l_{c_4}) \cap \boldsymbol{r} = \{r_1, r_2\}$. Furthermore, if the random bits are uniformly distributed in $\{0, 1\}$, then c_4 is both *uniformly distributed* and *secret independent* (Section 2.2).

2.2 Side-channel Attacks and Masking

We assume the adversary has access to the public input $\boldsymbol{p}$ and output $\boldsymbol{c}$, but not the secret $\boldsymbol{k}$ and random variable $\boldsymbol{r}$, of the program $\boldsymbol{c} \leftarrow P(\boldsymbol{p}, \boldsymbol{k})$. However, the adversary may have access to side-channel leaks that reveal the joint distribution of at most d intermediate computation results $c_1, \cdots c_d$ (e.g., via differential power analysis [39]). Under these assumptions, the goal of the adversary is to deduce information of $\boldsymbol{k}$. 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 transition-based model [5] can be used similarly [6].

Let $[n]$ denote the set $\{1, \cdots, n\}$ of natural numbers where $n \geq 1$. We call a set with d elements a *d-set*. Given values (p, k) for $(\boldsymbol{p}, \boldsymbol{k})$ and a d-set $\{c_1, \cdots, c_d\}$ of intermediate computation results, we use $\mathsf{D}_{p,k}(c_1, \cdots c_d)$ to denote their joint distribution induced by instantiating $\boldsymbol{p}$ and $\boldsymbol{k}$ with p and k, respectively. Formally, for each vector of values $v_1, \cdots, v_d$ in the probability space $\{0, 1\}^d$, we have $\mathsf{D}_{p,k}(c_1, \cdots c_d)(v_1, \cdots, v_d) =$

$$\frac{|\{r \in \{0,1\}^{|\boldsymbol{r}|} \mid v_1 = \mathbf{i}_1(\boldsymbol{p} = p, \boldsymbol{k} = k, \boldsymbol{r} = r), \cdots, v_d = \mathbf{i}_d(\boldsymbol{p} = p, \boldsymbol{k} = k, \boldsymbol{r} = r)\}|}{2^{|\boldsymbol{r}|}}.$$

Definition 1. *We say a d-set* $\{c_1, \cdots, c_d\}$ *of intermediate computation results is*

- uniformly distributed *if* $\mathsf{D}_{p,k}(c_1, \cdots, c_d)$ *is a uniform distribution for any* p *and* k.
- secret independent *if* $\mathsf{D}_{p,k}(c_1, \cdots, c_d) = \mathsf{D}_{p,k'}(c_1, \cdots, c_d)$ *for any* (p, k) *and* (p, 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. *A program P is* order-d perfectly masked *if every k-set* $\{c_1, \cdots, c_k\}$ *of P such that* $k \leq d$ *is secret independent. When P is (order-*1*) perfectly masked, we may simply say it is perfectly masked.*

To decide if P is order-d perfectly masked, it suffices to check if there exist a d-set and two pairs (p, k) and (p, k') such that $\mathsf{D}_{p,k}(c_1, \cdots, c_d) \neq \mathsf{D}_{p,k'}(c_1, \cdots, c_d)$. In this context, the main challenge is computing $\mathsf{D}_{p,k}(c_1, \cdots, c_d)$ which is essentially a *model-counting* (SAT#) problem. In the remainder of this paper, we focus on developing an efficient method for verifying (order-1) perfect masking, although our method can be extended to higher-order masking as well.

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 (structural information provided by $\texttt{supp}$ 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 $\texttt{semd}$ instead of $\texttt{supp}$ 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.

3 The Semantic Type Inference System

We first introduce our distribution types, which are inspired by prior work in [47,6,13], together with some auxiliary data structures; then, we present our inference rules.

3.1 The Type System

Let $\mathtt{T} = \{\mathtt{CST}, \mathtt{RUD}, \mathtt{SID}, \mathtt{NPM}, \mathtt{UKD}\}$ be the set of distribution types for intermediate computation results, where $[\![c]\!]$ denotes the type of $c \leftarrow \mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$. 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 secret independent, i.e., perfectly masked;
- $[\![c]\!] = \mathtt{NPM}$ means c is not perfectly masked and thus has leaks; and
- $[\![c]\!] = \mathtt{UKD}$ means c has an unknown distribution.

Definition 3. *Let* $\mathtt{unq} : N \rightarrow \boldsymbol{r}$ *and* $\mathtt{dom} : N \rightarrow \boldsymbol{r}$ *be two functions such that (i) for each terminal node* $l \in L_V$*, if* $\lambda_1(l) \in \boldsymbol{r}$*, then* $\mathtt{unq}(l) = \mathtt{dom}(l) = \lambda_1(l)$*; otherwise* $\mathtt{unq}(l) = \mathtt{dom}(l) = \mathtt{supp}(l) = \emptyset$*; and (ii) for each internal node* $l \in L$*, we have*

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

Both $\mathtt{unq}(l)$ and $\mathtt{dom}(l)$ are computable in time that is linear in $|P|$ [47]. Following the proofs in [47,6], it is easy to reach this observation: Given an intermediate computation result $c \leftarrow \mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ labeled by l, the following statements hold:

1. if $|\mathtt{dom}(l)| \neq \emptyset$, then $[\![c]\!] = \mathtt{RUD}$;
2. if $[\![c]\!] = \mathtt{RUD}$, then $[\![\neg c]\!] = \mathtt{RUD}$; if $[\![c]\!] = \mathtt{SID}$, then $[\![\neg c]\!] = \mathtt{SID}$.
3. if $r \notin \mathtt{semd}(l)$ for a random bit $r \in \boldsymbol{r}$, then $[\![r \oplus c]\!] = \mathtt{RUD}$;
4. for every $c' \leftarrow \mathbf{i}'(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ labeled by l', if $\mathtt{semd}(l) \cap \mathtt{semd}(l') = \emptyset$ and $[\![c]\!] = [\![c']\!] = \mathtt{SID}$, then $[\![c \circ c']\!] = \mathtt{SID}$.

Fig. 3 shows our type inference rules that concretize these observations. 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, we set $[\![l]\!] = \mathtt{UKD}$. When the context is clear, we may use $[\![l]\!]$ and $[\![c]\!]$ exchangeably for $\lambda_1(l) = c$. The correctness of these inference rules is obvious by definition.

Theorem 1. *For every intermediate computation result* $c \leftarrow \mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ *labeled by* l*,*
– *if* $[\![c]\!] = \mathtt{RUD}$*, then* c *is uniformly distributed, and hence perfectly masked;*
– *if* $[\![c]\!] = \mathtt{SID}$*, then* c *is guaranteed to be perfectly masked.*

$$\textsc{Leaf}_1\ \frac{\lambda_1(l) \in \boldsymbol{r}}{[\![l]\!] = \mathtt{RUD}} \qquad \textsc{Leaf}_2\ \frac{\lambda_1(l) \in \boldsymbol{p} \cup \boldsymbol{k}}{[\![l]\!] = \mathtt{UKD}} \qquad \textsc{Leaf}_3\ \frac{\lambda_1(l) \in \{0, 1\}}{[\![l]\!] = \mathtt{CST}}$$

$$\textsc{Xor-Rud}_1\ \frac{\lambda_2(l) = \oplus \quad [\![l.\mathtt{lft}]\!] = \mathtt{RUD} \quad \mathtt{dom}(l.\mathtt{lft}) \setminus \mathtt{semd}(l.\mathtt{rgt}) \neq \emptyset}{[\![l]\!] = \mathtt{RUD}} \qquad \textsc{Xor-Rud}_2\ \frac{\lambda_2(l) = \oplus \quad [\![l.\mathtt{rgt}]\!] = \mathtt{RUD} \quad \mathtt{dom}(l.\mathtt{rgt}) \setminus \mathtt{semd}(l.\mathtt{lft}) \neq \emptyset}{[\![l]\!] = \mathtt{RUD}}$$

$$\textsc{AO-Rud}_1\ \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{rgt}]\!] \notin \{\mathtt{UKD}, \mathtt{NPM}\} \quad [\![l.\mathtt{lft}]\!] = \mathtt{RUD} \quad \mathtt{semd}(l.\mathtt{lft}) \cap \mathtt{semd}(l.\mathtt{rgt}) = \emptyset}{[\![l]\!] = \mathtt{SID}}$$

$$\textsc{AO-Rud}_2\ \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{lft}]\!] \notin \{\mathtt{UKD}, \mathtt{NPM}\} \quad [\![l.\mathtt{rgt}]\!] = \mathtt{RUD} \quad \mathtt{semd}(l.\mathtt{rgt}) \cap \mathtt{semd}(l.\mathtt{lft}) = \emptyset}{[\![l]\!] = \mathtt{SID}}$$

$$\textsc{AO-Rud}_3\ \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\mathtt{lft}]\!] = [\![l.\mathtt{rgt}]\!] = \mathtt{RUD} \quad (\mathtt{dom}(l.\mathtt{lft}) \setminus \mathtt{semd}(l.\mathtt{rgt})) \cup (\mathtt{dom}(l.\mathtt{rgt}) \setminus \mathtt{semd}(l.\mathtt{lft})) \neq \emptyset}{[\![l]\!] = \mathtt{SID}}$$

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

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

Fig. 3. Our semantic type-inference rules. The NPM type is not yet used here; its inference rules will be added in Fig. 4 since they rely on the SMT solver based analyses.

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

Example 1. When using type inference with supp on the running example, 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 semd (for the second time), we have

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

3.2 Checking Semantic Independence

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

$$\Theta_s^{r=0}(c_0, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r} \setminus \{r\}) \wedge \Theta_s^{r=1}(c_1, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r} \setminus \{r\}) \wedge \Theta_s^{\neq}(c_0, c_1),$$

where $\Theta_s^{r=0}$ (resp. $\Theta_s^{r=1}$) encodes the relation $\mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ with r replaced by 0 (resp. 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 $r \notin \mathtt{semd}(l)$, r will be called the *don't care* variable [36]. Therefore, it is easy to see why the following theorem holds.

Theorem 2. *Φ_s 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 is linear in $|P|$.*

$$\text{CP-RUD}\;\frac{[\![c_1,\cdots,c_k]\!]=\texttt{RUD}\quad [\![c_{k+1}]\!]=\texttt{RUD}\quad \texttt{semd}(c_1,\cdots,c_k)\cap\texttt{semd}(c_{k+1})=\emptyset}{[\![c_1,\cdots,c_{k+1}]\!]=\texttt{RUD}}$$

$$\text{CP-SID}_1\;\frac{[\![c_1,\cdots,c_k]\!],[\![c_{k+1}]\!]\in\{\texttt{SID},\texttt{RUD}\}\quad [\![c_{k+1}]\!]\neq[\![c_1,\cdots,c_k]\!]\quad \texttt{semd}(c_1,\cdots,c_k)\cap\texttt{semd}(c_{k+1})=\emptyset}{[\![c_1,\cdots,c_{k+1}]\!]=\texttt{SID}}$$

$$\text{CP-SID}_2\;\frac{[\![c_1,\cdots,c_k]\!]=\texttt{RUD}\quad [\![c_{k+1}]\!]=\texttt{RUD}\quad (\texttt{dom}(c_1,\cdots,c_k)\setminus\texttt{semd}(c_{k+1}))\cap(\texttt{dom}(c_{k+1})\setminus\texttt{semd}(c_1,\cdots,c_k))\neq\emptyset}{[\![c_1,\cdots,c_{k+1}]\!]=\texttt{SID}}$$

$$\text{CP-UKD}\;\frac{\text{no-rule is appliable at }\{c_1,\cdots,c_{k+1}\}}{[\![c_1,\cdots,c_{k+1}]\!]=\texttt{UKD}}$$

Fig. 4. Our composition rules for handling *sets* of intermediate computation results.

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 k-set $\{c_1,\cdots,c_k\}$ of intermediate computation results such that $k \le d$, the joint distribution is either randomized to uniform distribution (RUD) or secret independent (SID).

To tackle this problem, we lift supp, semd, unq, and dom to *sets* of computation results as follows: for each k-set $\{c_1,\cdots,c_k\}$,

- $\texttt{supp}(c_1,\cdots,c_k)=\bigcup_{i\in[k]}\texttt{supp}(c_i)$;
- $\texttt{semd}(c_1,\cdots,c_k)=\bigcup_{i\in[k]}\texttt{semd}(c_i)$;
- $\texttt{unq}(c_1,\cdots,c_k)=(\bigcup_{i\in[k]}\texttt{unq}(c_i))\setminus\bigcup_{i,j\in[k]}(\texttt{supp}(c_i)\cap\texttt{supp}(c_j))$; and
- $\texttt{dom}(c_1,\cdots,c_k)=(\bigcup_{i\in[k]}\texttt{dom}(c_i))\cap\texttt{unq}(c_1,\cdots,c_k)$.

Our inference rules are extended by adding the composition rules shown in Fig. 4.

Theorem 3. *For every k-set $\{c_1,\cdots,c_k\}$ of intermediate computations results,*

- *if $[\![c_1,\cdots,c_k]\!]=\texttt{RUD}$, then $\{c_1,\cdots,c_k\}$ is guaranteed to be uniformly distributed, and hence perfectly masked;*
- *if $[\![c_1,\cdots,c_k]\!]=\texttt{SID}$, then $\{c_1,\cdots,c_k\}$ is guaranteed to be perfectly masked.*

We remark that the semd function in these composition rules could also be safely replaced by the supp function, just as before. Furthermore, to more efficiently verify that program P is perfect masked against order-d attacks, we can incrementally apply the type inference for each k-set, where $k=1,2,\ldots,d$.

4 The Gradual Refinement Approach

In this section, we present our method for gradually refining the type inference 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 $c \leftarrow \mathbf{i}(\boldsymbol{p},\boldsymbol{k},\boldsymbol{r})$, the verification of perfect masking (Definition 2) can be reduced to the *satisfiability* of the logical formula (Ψ) defined as follows:

$$\exists \boldsymbol{p}.\exists \boldsymbol{k}.\exists \boldsymbol{k}'.(\sum_{\boldsymbol{v}_r\in\{0,1\}^{|r|}} \mathbf{i}(\boldsymbol{p},\boldsymbol{k},\boldsymbol{v}_r) \neq \sum_{\boldsymbol{v}_r\in\{0,1\}^{|r|}} \mathbf{i}(\boldsymbol{p},\boldsymbol{k}',\boldsymbol{v}_r)).$$

Intuitively, given values $(\boldsymbol{v}_p, \boldsymbol{v}_k)$ of $(\boldsymbol{p}, \boldsymbol{k})$, $count = \sum_{\boldsymbol{v}_r\in\{0,1\}^{|r|}} \mathbf{i}(\boldsymbol{v}_p, \boldsymbol{v}_k, \boldsymbol{v}_r)$ denotes the number of assignments of the random variable $\boldsymbol{r}$ under which $\mathbf{i}(\boldsymbol{v}_p, \boldsymbol{v}_k, \boldsymbol{r})$ is evaluated to logical 1. When random bits in $\boldsymbol{r}$ are uniformly distributed in the domain $\{0, 1\}$, $\frac{count}{2^{|r|}}$ is the probability of $\mathbf{i}(\boldsymbol{v}_p, \boldsymbol{v}_k, \boldsymbol{r})$ being logical 1 for the given pair $(\boldsymbol{v}_p, \boldsymbol{v}_k)$. Therefore, Ψ is unsatisfiable if and only if c is perfectly masked.

Following Eldib et al. [27,26], we encode the formula Ψ as a quantifier-free first-order logic formula to be solved by an off-the-shelf SMT solver (e.g., Z3):

$$(\bigwedge_{r=0}^{2^{|r|}-1} \Theta_k^r) \wedge (\bigwedge_{r=0}^{2^{|r|}-1} \Theta_{k'}^r) \wedge \Theta_{b2i} \wedge \Theta_{\neq}$$

- Θ_k^v (resp. $\Theta_{k'}^v$) for each $r \in \{0, \cdots, 2^{|\boldsymbol{r}|} - 1\}$: encodes a copy of the input-output relation of $\mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, r)$ (resp. $\mathbf{i}(\boldsymbol{p}, \boldsymbol{k}', r)$) by replacing $\boldsymbol{r}$ with concrete values r. There are $2^{|\boldsymbol{r}|}$ distinct copies, but share the same plaintext $\boldsymbol{p}$.
- 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 $\boldsymbol{k}$ and $\boldsymbol{k}'$, differ.

Example 2. In the running example, for instance, verifying whether node c_4 is perfectly masked requires the SMT-based analysis. For brevity, we omit the detailed logical formula while pointing out that, by invoking the SMT solver six times, one can get the following result: $[\![c_1]\!] = [\![c_2]\!] = [\![c_3]\!] = [\![c_4]\!] = [\![c_5]\!] = [\![c_6]\!] = \mathtt{SID}$.

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 [27,26] were proposed to reduce the formula size using partitioning and heuristic reduction.

Incremental SMT-based algorithm. Given a computation $c \leftarrow \mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ 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 $\mathtt{dom}(l_s) \cap \mathtt{unq}(l) \neq \emptyset$. 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 by a maximal cut-point with a random variable from $\mathtt{dom}(l_s) \cap \mathtt{unq}(l)$. Then, $\widehat{T}$ is SID iff T is SID.

The main observation is that: if l_s is a cut-point, there is a random variable $r \in \mathtt{dom}(l_s) \cap \mathtt{unq}(l)$, which implies $\lambda_1(l_s)$ is RUD. Here, $r \in \mathtt{unq}(l)$ implies $\lambda_1(l_s)$ can be seen as a *fresh* random variable when we evaluate l. Consider the node c_3 in our running example: , it is easy to see $r_1 \in \mathtt{dom}(c_2) \cap \mathtt{unq}(c_3)$. 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 [27,26] can be used to simplify SMT solving. (1) When constructing formula Φ of c, all random variables in $\mathtt{supp}(l) \setminus \mathtt{semd}(l)$, which are *don't cares*, can be replaced by constant 1 or 0. (2) The No-Key and Sid rules in Fig. 3 with the supp function are used to skip some checks by SMT.

Example 3. When applying incremental SMT-based approach to our running example, c_1 has to be decided by SMT, but c_2 is skipped due to No-Key rule.

As for c_3, since $r_1 \in \mathtt{dom}(c_2) \cap \mathtt{unq}(c_3)$, 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

$$\text{AO-NPM}_1 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{rgt}]\!] = \texttt{NPM} \quad [\![l.\texttt{lft}]\!] = \texttt{RUD} \quad \texttt{semd}(l.\texttt{lft}) \cap \texttt{semd}(l.\texttt{rgt}) = \emptyset}{[\![l]\!] = \texttt{NPM}} \qquad \text{AO-NPM}_2 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{lft}]\!] = \texttt{NPM} \quad [\![l.\texttt{rgt}]\!] = \texttt{RUD} \quad \texttt{semd}(l.\texttt{rgt}) \cap \texttt{semd}(l.\texttt{lft}) = \emptyset}{[\![l]\!] = \texttt{NPM}}$$

$$\text{AO-NPM}_3 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{rgt}]\!] = \texttt{NPM} \quad [\![l.\texttt{lft}]\!] = \texttt{RUD} \quad \texttt{dom}(l.\texttt{lft}) \setminus \texttt{semd}(l.\texttt{rgt}) \neq \emptyset}{[\![l]\!] = \texttt{NPM}} \qquad \text{AO-NPM}_4 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad [\![l.\texttt{lft}]\!] = \texttt{NPM} \quad [\![l.\texttt{rgt}]\!] = \texttt{RUD} \quad \texttt{dom}(l.\texttt{rgt}) \setminus \texttt{semd}(l.\texttt{lft}) \neq \emptyset}{[\![l]\!] = \texttt{NPM}}$$

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

Fig. 5. Complementary rules used during refinement of the type inference (Fig. 3).

it is skipped by the SID rule with supp. 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 formula Φ.

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., $[\![l]\!]$ = 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, the type of l can be refined to $[\![l]\!]$ = SID; feeding it back to the type system allows us to infer more types for nodes that structurally depend on l.

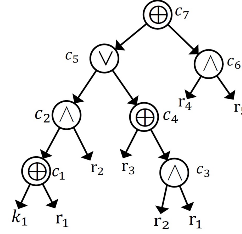

Fig. 6. Example for feeding back.

On the other hand, if SMT-based analysis shows l is not perfectly masked, the type of l can be refined to $[\![l]\!]$ = 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 Fig. 5. When they are added to the type system outlined in Fig. 3, more NPM type nodes will be deduced, which allows our method to skip the (more expensive) checking of NPM using SMT.

Example 4. Consider the example DDG in Figure 6. By applying the original type inference approach with either supp or semd, we have

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

In contrast, by applying SMT-based analysis to c_5, we can deduce $[\![c_5]\!]$ = SID. Feeding $[\![c_5]\!]$ = 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 $[\![c_7]\!]$ = SID. Without refinement, this was not possible.

Algorithm 1: Function SCInfer($P, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r}, \pi$)

```
1  Function SCInfer(P, p, k, r, π)
2    foreach l ∈ N in a topological order do
3      if l is a leaf then π(l) := [[l]];
4      else
5        TypeInfer(l, P, p, k, r, π, supp);
6        if π(l) = UKD then
7          let P̂ be the simplified tree of the subtree rooted by l in P;
8          TypeInfer(l, P̂, p, k, r, π, semd);
9          if π(l) = UKD then
10           res:=CheckBySMT(P̂, p, k, r);
11           if res=Not-Perfectly-Masked then π(l) := NPM;
12           else if res=Perfectly-Masked then π(l) := SID;
13           else π(l) := UKD;
```

4.3 The Overall Algorithm

Having presented all the components, we now present the overall procedure, 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 ($\boldsymbol{p}$), secret ($\boldsymbol{k}$), random ($\boldsymbol{r}$) variables and an empty map π, it invokes SCInfer($P, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r}, \pi$) 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 Fig. 3 and Fig. 5, where the parameter f can be either supp or semd.

SCInfer first deduces the type of each node $l \in N$ by invoking TypeInfer with f = supp. Once a node l is annotated as UKD, a simplified subtree $\widehat{P}$ of the subtree rooted at l is constructed. Next, TypeInfer with f = semd is invoked to resolve the UKD node in $\widehat{P}$. If $\pi(l)$ becomes non-UKD afterward, TypeInfer with f = supp is invoked again to quickly deduce the types of the fan-out nodes in P. But if $\pi(l)$ remains UKD, 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. *For every intermediate computation result $c \leftarrow \mathbf{i}(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r})$ labeled by l, our method in* SCInfer *guarantees to return sound and complete results:*

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

If timeout or memory out is used to bound the execution of the SMT solver, it is also possible that $\pi(l)$ = 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, Algorithm 1 degenerates to a *sound* type system that is both fast and potentially accurate.

Algorithm 2: Procedure TypeInfer$(l, P, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r}, \pi, f)$

1 **Procedure** TypeInfer$(l, P, \boldsymbol{p}, \boldsymbol{k}, \boldsymbol{r}, \pi, f)$
2   **if** $\lambda_2(l) = \neg$ **then** $\pi(l) := \pi(l.\mathtt{lft})$;
3   **else if** $\lambda_2(l) = \oplus$ **then**
4   **if** $\pi(l.\mathtt{lft}) = \mathtt{RUD} \wedge \mathsf{dom}(l.\mathtt{lft}) \setminus f(l.\mathtt{rgt}) \neq \emptyset$ **then** $\pi(l) := \mathtt{RUD}$;
5   **else if** $\pi(l.\mathtt{rgt}) = \mathtt{RUD} \wedge \mathsf{dom}(l.\mathtt{rgt}) \setminus f(l.\mathtt{lft}) \neq \emptyset$ **then** $\pi(l) := \mathtt{RUD}$;
6   **else if** $\pi(l.\mathtt{rgt}) = \pi(l.\mathtt{lft}) = \mathtt{SID} \wedge f(l.\mathtt{lft}) \cap f(l.\mathtt{rgt}) \cap \boldsymbol{r} = \emptyset$ **then**
7       $\pi(l) := \mathtt{SID}$
8   **else if** $\mathsf{supp}(l) \cap \boldsymbol{k} = \emptyset$ **then** $\pi(l) := \mathtt{SID}$;
9   **else** $\pi(l) := \mathtt{UKD}$;
10   **else**
11   **if** $\left(\begin{array}{c}\big((\pi(l.\mathtt{lft}) = \mathtt{RUD} \wedge \pi(l.\mathtt{rgt}) \notin \{\mathtt{UKD}, \mathtt{NPM}\}) \vee \\ (\pi(l.\mathtt{rgt}) = \mathtt{RUD} \wedge \pi(l.\mathtt{lft}) \notin \{\mathtt{UKD}, \mathtt{NPM}\})\big) \\ \wedge f(l.\mathtt{lft}) \cap f(l.\mathtt{rgt}) \cap \boldsymbol{r} = \emptyset\end{array}\right)$ **then** $\pi(l) := \mathtt{SID}$;
12   **else if** $\left(\begin{array}{c}(\mathsf{dom}(l.\mathtt{rgt}) \setminus f(l.\mathtt{lft})) \cup (\mathsf{dom}(l.\mathtt{lft}) \setminus f(l.\mathtt{rgt})) \neq \emptyset \\ \wedge \pi(l.\mathtt{lft}) = \mathtt{RUD} \wedge \pi(l.\mathtt{rgt}) = \mathtt{RUD}\end{array}\right)$ **then**
13       $\pi(l) := \mathtt{SID}$
14   **else if** $\left(\begin{array}{c}\big((\pi(l.\mathtt{lft}) = \mathtt{RUD} \wedge \pi(l.\mathtt{rgt}) = \mathtt{NPM}) \vee \\ (\pi(l.\mathtt{rgt}) = \mathtt{RUD} \wedge \pi(l.\mathtt{lft}) = \mathtt{NPM})\big) \\ \wedge f(l.\mathtt{lft}) \cap f(l.\mathtt{rgt}) \cap \boldsymbol{r} = \emptyset\end{array}\right)$ **then** $\pi(l) := \mathtt{NPM}$;
15   **else if** $\left(\begin{array}{c}(\pi(l.\mathtt{lft}) = \mathtt{RUD} \wedge \pi(l.\mathtt{rgt}) = \mathtt{NPM} \wedge \mathsf{dom}(l.\mathtt{lft}) \setminus f(l.\mathtt{rgt}) \neq \emptyset) \vee \\ (\pi(l.\mathtt{rgt}) = \mathtt{RUD} \wedge \pi(l.\mathtt{lft}) = \mathtt{NPM} \wedge \mathsf{dom}(l.\mathtt{rgt}) \setminus f(l.\mathtt{lft}) \neq \emptyset)\end{array}\right)$ **then**
16       $\pi(l) := \mathtt{NPM}$
17   **else if** $(\pi(l.\mathtt{lft}) = \pi(l.\mathtt{rgt}) = \mathtt{SID}) \wedge f(l.\mathtt{lft}) \cap f(l.\mathtt{rgt}) \cap \boldsymbol{r} = \emptyset$ **then**
18       $\pi(l) := \mathtt{SID}$
19   **else if** $\mathsf{supp}(l) \cap \boldsymbol{k} = \emptyset$ **then** $\pi(l) := \mathtt{SID}$;
20   **else** $\pi(l) := \mathtt{UKD}$;

5 Experiments

We have implemented our method in a verification tool named SCInfer, which uses Z3 [23] as the underlying SMT solver. We also implemented the syntactic type inference approach [47] and the incremental SMT-based approach [27,26] in the same tool for experimental comparison purposes. We conducted experiments on publicly available cryptographic software implementations, including fragments of AES and MAC-Keccak [27,26]. 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 (1) SCInfer is significantly more accurate than prior syntactic type inference method [47]; indeed, it solved tens of thousand of UKD cases reported by the prior technique; (2) SCInfer is at least twice faster than prior SMT-based verification method [27,26] on the large programs while maintaining the same accuracy; for example, SCInfer verified the benchmark named P12 in a few seconds whereas the prior SMT-based method took more than an hour.

Table 1. Benchmark statistics.

Name	Description	♯Loc	♯Nodes	$\|\boldsymbol{k}\|$	$\|\boldsymbol{p}\|$	$\|\boldsymbol{r}\|$
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

5.1 Benchmarks

Table 1 shows the detailed statistics of the benchmarks, including seventeen 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 DDG nodes 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 computation is far less than the one of the program. All these programs are transformed into Boolean programs 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 [27,26].

In particular, P1-P5 are masking examples originated from [10], P6-P7 are originated from [15], P8-P9 are the MAC-Keccak computation reordered examples originated from [11], P10-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.

5.2 Experimental Results

We compared the performance of SCInfer, 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 3-4 show the results of the purely syntactic type inference method, including the number of nodes inferred as `UKD` type and the time in seconds. Columns 5-7 (resp. Columns 8-10) show the results of the incremental SMT-based method (resp. our method SCInfer), including the number of leaky nodes (`NPM` type), the number of nodes actually checked by SMT, and the time.

Table 2. Experimental results: comparison of three approaches.

Name	Masked	Syn. Infer [47]		SMT App [27,26]			SCInfer		
		UKD	Time	NPM	By SMT	Time	NPM	By SMT	Time
P1	No	16	≈0s	16	16	0.39s	16	16	0.39s
P2	No	8	≈0s	8	8	0.28s	8	8	0.57s
P3	Yes	0	≈0s	0	0	≈0s	0	0	≈0s
P4	Yes	**3**	≈0s	0	**3**	0.16s	**0**	**0**	0.06s
P5	Yes	**3**	≈0s	0	**3**	0.15s	**0**	**2**	0.25s
P6	No	2	≈0s	2	2	0.11s	2	2	0.16s
P7	No	2	0.01s	1	2	0.11s	1	1	0.26s
P8	No	3	≈0s	3	3	0.15s	3	3	0.29s
P9	No	2	≈0s	2	2	0.11s	2	2	0.23s
P10	No	3	≈0s	1	2	0.15s	1	2	0.34s
P11	No	4	≈0s	1	3	0.2s	1	3	0.5s
P12	Yes	0	**1m 5s**	0	0	92m 8s	0	0	**3.8s**
P13	No	4800	1m 11s	4800	4800	95m 30s	4800	4800	47m 8s
P14	No	3200	1m 11s	3200	3200	118m 1s	3200	3200	53m 40s
P15	No	**3200**	1m 21s	1600	3200	127m 45s	**1600**	**3200**	69m 6s
P16	No	4800	1m 13s	4800	4800	123m 54s	4800	4800	61m 15s
P17	No	17600	1m 14s	17600	16000	336m 51s	**17600**	**12800**	121m 28s

Compared with syntactic type inference method, our approach is significantly more accurate (e.g., see P4, P5 and P15). 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 SCInfer has to resolve the UKD nodes reported by syntactic inference by SMT. However, it is interesting to note that, on the perfectly masked large program (P12), our method is faster.

Moreover, the UKD type 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 type 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) allows our type system to improve its accuracy, which turns the third UKD node to SID.

Finally, compared with the original SMT-based approach, our method is at least 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-P5, and P17). In particular, there are 3200 UKD type nodes in P17, which are refined into NPM type by our new inference rules (cf. Fig. 5), and thus avoid the more expensive SMT calls.

To sum up, results of our experiments show that: SCInfer 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.

5.3 Detailed Statistics

Table 3 shows the more detailed statistics of our approach. Specifically, Columns 2-5 show the number of nodes in each distribution type deduced by our method. Column 6 shows the number of nodes actually checked by SMT, together with the time shown in Column 9. Column 7 shows the time spent on computing the semd function, which

Table 3. Detailed statistics of our new method.

Name	SCInfer								
Name	Nodes					Time			
	RUD	SID	CST	NPM	SMT	semd	Don't care	SMT	Total
P1	16	0	0	16	16	≈0s	≈0s	0.39s	0.39s
P2	16	0	0	8	8	0.27s	0.14s	0.16s	0.57s
P3	6	0	0	0	0	≈0s	≈0s	≈0s	≈0s
P4	6	0	0	0	0	≈0s	≈0s	≈0s	0.06s
P5	6	2	0	0	2	0.08s	0.05s	0.05s	0.25s
P6	4	3	0	2	2	0.05s	0.07s	0.04s	0.16s
P7	5	5	0	1	1	0.14s	0.09s	0.03s	0.26s
P8	11	4	0	3	3	0.14s	0.09s	0.06s	0.29s
P9	12	4	0	2	2	0.13s	0.07s	0.03s	0.23s
P10	20	6	1	1	2	0.15s	0.14s	0.05s	0.34s
P11	19	7	1	1	3	0.23s	0.2s	0.07s	0.5s
P12	190400	6400	0	0	0	≈0s	≈0s	≈0s	3.8s
P13	185600	6400	0	4800	4800	29m 33s	16m 5s	1m 25s	47m 8s
P14	187200	6400	0	3200	3200	26m 58s	25m 26s	1m 53s	53m 40s
P15	188800	8000	0	1600	3200	33m 30s	33m 55s	1m 35s	69m 6s
P16	185600	6400	0	4800	4800	26m 41s	32m 55s	1m 32s	61m 15s
P17	185600	1600	0	17600	12800	33m 25s	83m 59s	3m 57s	121m 28s

solves the SAT problem. Column 8 shows the time spent on computing the don't care variables. The last column shows the total time taken by SCInfer.

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. It explains why our new method is more efficient than the original SMT-based approach. Indeed, the original SMT-based approach spent a large amount of time on the static analysis part, which does code partitioning and applies the heuristic rules (cf. Section 4.1), whereas our method spent more time on computing the semd function.

Column 4 shows that, at least in these benchmark programs, Boolean constants are rare. Columns 5-6 show that, if our refined type system fails to prove perfect masking, it is usually not perfectly masked. Columns 7-9 show that, in our integrated method, most of the time is actually used to compute semd and don't care variables (SAT), while the time taken by the SMT solver to conduct model counting (SAT#) is relatively small.

6 Related Work

Many masking countermeasures [41,34,37,15,46,52,17,51,43,48,50] have been published over the years: although they differ in adversary models, cryptographic algorithms and compactness, a common problem is the lack of efficient tools to formally prove their correctness [21,22]. Our work aims to bridge the gap. It differs from simulation-based techniques [33,3,53] which aim to detect leaks only as opposed to prove their absence. It also differs from techniques designed for other types of side channels such as timing [38,2], fault [12,29] and cache [35,40,24], or computing security bounds for probabilistic countermeasures against remote attacks [45].

Although some verification tools have been developed for this application [10,27,26,6,47,13,7,20,14], they are either fast but inaccurate (e.g., type-inference techniques) or accurate but slow (e.g., model-counting techniques). For example, Bayrak et al. [10] developed a leak detector that checks if a computation result is

logically dependent of the secret and, at the same time, *logically* independent of any random variable. It is fast but not accurate in that many leaky nodes could be incorrectly proved [27,26]. In contrast, the model-counting based method proposed by Eldib et al. [27,26,28] is accurate, but also significantly less scalable because the size of logical formulas they need to build are exponential in the number of random variables. Moreover, for higher-order masking, their method is still not complete.

Our gradual refinement of a set of semantic type inference rules were inspired by recent work on proving probabilistic non-interference [6,47], which exploit the unique characteristics of invertible operations. Similar ideas were explored in [7,20,14] as well. However, these prior techniques differ significantly from our method because their type-inference rules are syntactic and fixed, whereas ours are semantic and refined based on SMT solver based analysis (SAT and SAT#). In terms of accuracy, numerous unknowns occurred in the experimental results of [47] and two obviously perfect masking cases were not proved in [6]. Finally, although higher-order masking were addressed by prior techniques [13], they were limited to linear operations, whereas our method can handle both first-order and higher-order masking with non-linear operations.

An alternative way to address the model-counting problem [19,18,4,32] is to use satisfiability modulo counting, which is a generalization of the satisfiability problem of SMT extended with counting constraints [31]. Toward this end, Fredrikson and Jha [31] have developed an efficient decision procedure for linear integer arithmetic (LIA) based on Barvinok's algorithm [8] and also applied their approach to differential privacy.

Another related line of research is automatically synthesizing countermeasures [9,44,1,25,7,16,54] as opposed to verifying them. While methods in [9,44,1,7] rely on compiler-like pattern matching, the ones in [25,16,54] use inductive program synthesis based on the SMT approach. These emerging techniques, however, are orthogonal to our work reported in this paper. It would be interesting to investigate whether our approach could aid in the synthesis of masking countermeasures.

7 Conclusions and Future Work

We have presented a refinement based method for proving that a piece of cryptographic software code is free of power side-channel leaks. 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 method and demonstrated its efficiency and effectiveness on cryptographic benchmarks. Our results show that it outperforms state-of-the-art techniques in terms of both efficiency and accuracy.

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, e.g., using satisfiability modulo counting [31], and investigate the synthesis of masking countermeasures based on our new verification method.

References

1. Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*,

pages 77–82, 2012.
2. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.
3. Victor Arribas, Svetla Nikova, and Vincent Rijmen. Vermi: Verification tool for masked implementations. *IACR Cryptology ePrint Archive*, page 1227, 2017.
4. Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272, 2015.
5. Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications (CARDIS)*, pages 64–81, 2014.
6. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *International Conference on the Theory and Applications of Cryptographic (EUROCRYPT)*, pages 457–485, 2015.
7. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *ACM Conference on Computer and Communications Security*, pages 116–129, 2016.
8. Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.
9. Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 230–235, 2011.
10. Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–310, 2013.
11. Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. `https://keccak.team/files/Keccak-implementation-3.2.pdf`, 2013.
12. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 513–525, 1997.
13. Elia Bisi, Filippo Melzani, and Vittorio Zaccaria. Symbolic analysis of higher-order side channel countermeasures. *IEEE Trans. Computers*, 66(6):1099–1105, 2017.
14. Roderick Bloem, Hannes Gross, Rinat Iusupov, Bettina Konighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. *IACR Cryptology ePrint Archive*, page 897, 2017.
15. Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. In *International Workshop on Selected Areas in Cryptography*, pages 69–83. Springer, 2004.
16. Arthur Blot, Masaki Yamamoto, and Tachio Terauchi. Compositional synthesis of leakage resilient programs. In *International Conference on Principles of Security and Trust*, pages 277–297, 2017.
17. D. Canright and Lejla Batina. A very compact "perfectly masked" s-box for AES. In *International Conference on Applied Cryptography and Network Security*, pages 446–459, 2008.
18. Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *AAAI Conference on Artificial Intelligence*, pages 1722–1730, 2014.

19. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*, pages 200–216, 2013.
20. Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. *IACR Cryptology ePrint Archive*, page 879, 2017.
21. Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side channel cryptanalysis of a higher order masking scheme. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 28–44, 2007.
22. Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In *International Workshop on Fast Software Encryption*, pages 410–424, 2013.
23. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.
24. Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security Symposium*, pages 431–446, 2013.
25. Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*, pages 114–130, 2014.
26. Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology*, 24(2):11, 2014.
27. Hassan Eldib, Chao Wang, and Patrick Schaumont. Smt-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 62–77, 2014.
28. Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*, pages 209:1–6, 2014.
29. Hassan Eldib, Meng Wu, and Chao Wang. Synthesis of fault-attack countermeasures for cryptographic circuits. In *International Conference on Computer Aided Verification*, pages 343–363, 2016.
30. Christophe Clavier et al. Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. *J. Cryptographic Engineering*, 4(4):259–274, 2014.
31. Matthew Fredrikson and Somesh Jha. Satisfiability modulo counting: a new approach for analyzing privacy properties. In *ACM/IEEE Symposium on Logic in Computer Science*, pages 42:1–42:10, 2014.
32. Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. Maximum model counting. In *AAAI Conference on Artificial Intelligence*, pages 3885–3892, 2017.
33. Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.
34. Louis Goubin. A sound method for switching between boolean and arithmetic masking. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15, 2001.
35. Philipp Grabher, Johann Großschädl, and Dan Page. Cryptographic side-channels from low-power cache memory. In *IMA International Conference, Cirencester on Cryptography and Coding*, pages 170–184, 2007.
36. Gary D. Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. Kluwer, 1996.
37. Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 463–481, 2003.

38. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 104–113, 1996.
39. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 388–397, 1999.
40. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*, pages 564–580, 2012.
41. Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In *International Workshop on Fast Software Encryption*, pages 150–164, 2000.
42. Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In *ACM Conference on Computer and Communications Security*, pages 111–124, 2011.
43. Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 69–88, 2011.
44. Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 58–75, 2012.
45. Martín Ochoa, Sebastian Banescu, Cynthia Disenfeld, Gilles Barthe, and Vijay Ganesh. Reasoning about probabilistic defense mechanisms against remote attacks. In *IEEE European Symposium on Security and Privacy, EuroS&P*, pages 499–513, 2017.
46. Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES s-box. In *International Workshop on Fast Software Encryption*, pages 413–423, 2005.
47. Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *Security Proofs for Embedded Systems*, 2017.
48. Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 142–159, 2013.
49. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *International Conference on Research in Smart Cards (E-smart)*, pages 200–210, 2001.
50. Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 764–783, 2015.
51. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427, 2010.
52. Kai Schramm and Christof Paar. Higher order masking of the AES. In *Proceedings of the RSA Conference on Topics in Cryptology (CT-RSA)*, pages 208–225, 2006.
53. François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. *IACR Cryptology ePrint Archive*, 2017:138, 2017.
54. Chao Wang and Patrick Schaumont. Security by compilation: an automated approach to comprehensive side-channel resistance. *SIGLOG News*, 4(2):76–89, 2017.

A Appendix

A.1 SMT Encoding of the Running Example

Example 5. Consider the intermediate computation result c_4 in the running example, by instantiating (r_1, r_2) to values from $\{0,1\}^2$, we can get the SMT encoding Φ, where the four components is given below:

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

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

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

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