



# SCINFER: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks

Jun Zhang<sup>1</sup>, Pengfei Gao<sup>1</sup>, Fu Song<sup>1</sup>(✉),  
and Chao Wang<sup>2</sup>

<sup>1</sup> ShanghaiTech University, Shanghai, China  
songfu@shanghaitech.edu.cn

<sup>2</sup> 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 [30, 42].

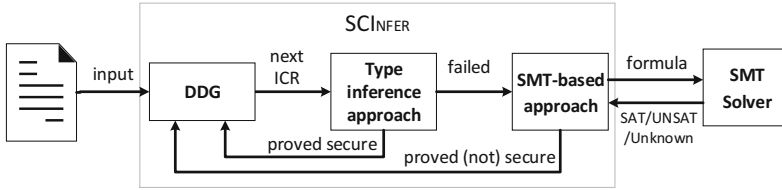
---

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.

© The Author(s) 2018

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10982, pp. 157–177, 2018.

[https://doi.org/10.1007/978-3-319-96142-2\\_12](https://doi.org/10.1007/978-3-319-96142-2_12)



**Fig. 1.** Overview of SCINFER, where “ICR” denotes the intermediate computation result.

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) [15,37,51,52], 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.

Previously, there are two types of verification methods for masking countermeasures [54]: one is type inference based [10,44] and the other is model counting based [26,27]. Type inference based methods [10,44] 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 [26,27] 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 [6,13,47]. 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. Figure 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 (Sect. 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 [26,27], 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 Sect. 2, we present our semantic type inference system in Sect. 3 and our refinement method in Sect. 4. Then, we present our experimental results in Sect. 5 and comparison with related work in Sect. 6. We give our conclusions in Sect. 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 [15,26,27], we assume that the program  $P$  implements a cryptographic function, e.g.,  $c \leftarrow P(p, k)$  where  $p$  is the plaintext,  $k$  is the secret key and  $c$  is the ciphertext. Inside  $P$ , random variable  $r$  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 [26,27] 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  $k$  (resp.  $r$ ) be the set of secret (resp. random) bits,  $p$  the public bits, and  $c$  the variables storing intermediate results. Thus, the set of variables is  $V = k \cup r \cup p \cup c$ . 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  $c_1 \leftarrow \mathbf{i}_1(p, k, r); \dots; c_n \leftarrow \mathbf{i}_n(p, k, r)$

where, for each  $1 \leq i \leq n$ , the value of  $\mathbf{i}_i$  is expressed in terms of  $p$ ,  $k$  and  $r$ . Each random bit in  $r$  is uniformly distributed in  $\{0, 1\}$ ; the sole purpose of using them in  $P$  is to ensure that  $c_1, \dots, c_n$  are statistically independent of the secret  $k$ .

**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  $\lambda$  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  $v \in k \cup r \cup p \cup \{0, 1\}$ .
- $E \subseteq N \times N$  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  $l'$ ;
- $\lambda$  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.

Figure 2 shows an example where  $k = \{k\}$ ,  $r = \{r_1, r_2, r_3\}$ ,  $c = \{c_1, c_2, c_3, c_4, c_5, c_6\}$  and  $p = \emptyset$ . 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

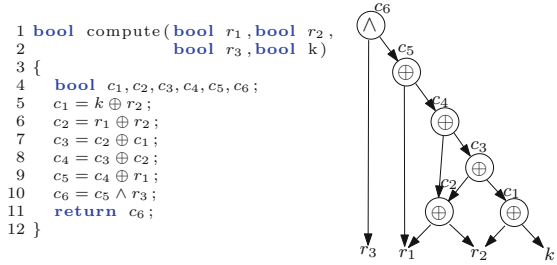


Fig. 2. An example for masking countermeasure.

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

Let  $\text{supp} : N \rightarrow \mathbf{k} \cup \mathbf{r} \cup \mathbf{p}$  be a function mapping each node  $l$  to its support variables. That is,  $\text{supp}(l) = \emptyset$  if  $\lambda_1(l) \in \{0, 1\}$ ;  $\text{supp}(l) = \{x\}$  if  $\lambda_1(l) = x \in \mathbf{k} \cup \mathbf{r} \cup \mathbf{p}$ ; and  $\text{supp}(l) = \text{supp}(l.\text{lft}) \cup \text{supp}(l.\text{rgt})$  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,  $\pi_1$  and  $\pi_2$ , 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  $\pi_1$  and  $\pi_2$ .

Let  $\text{semd} : N \rightarrow \mathbf{r}$  be a function such that  $\text{semd}(l)$  denotes the set of *random variables* upon which the expression  $e$  of  $l$  semantically depends. Thus,  $\text{semd}(l) \subseteq \text{supp}(l)$ ; and for each  $r \in \text{supp}(l) \setminus \text{semd}(l)$ , we know  $\lambda_1(l)$  is semantically independent of  $r$ . More importantly, there is often a gap between  $\text{supp}(l) \cap \mathbf{r}$  and  $\text{semd}(l)$ , namely  $\text{semd}(l) \subseteq \text{supp}(l) \cap \mathbf{r}$ , 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  $\text{supp}(l_{c_4}) = \{r_1, r_2, k\}$ ,  $\text{semd}(l_{c_4}) = \{r_2\}$ , and  $\text{supp}(l_{c_4}) \cap \mathbf{r} = \{r_1, r_2\}$ . Furthermore, if the random bits are uniformly distributed in  $\{0, 1\}$ , then  $c_4$  is both *uniformly distributed* and *secret independent* (Sect. 2.2).

## 2.2 Side-Channel Attacks and Masking

We assume the adversary has access to the public input  $\mathbf{p}$  and output  $\mathbf{c}$ , but not the secret  $\mathbf{k}$  and random variable  $\mathbf{r}$ , of the program  $\mathbf{c} \leftarrow P(\mathbf{p}, \mathbf{k})$ . However, the adversary may have access to side-channel leaks that reveal the joint distribution of at most  $d$  intermediate computation results  $c_1, \dots, c_d$  (e.g., via differential power analysis [39]). Under these assumptions, the goal of the adversary is to deduce information of  $\mathbf{k}$ . 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  $\{1, \dots, n\}$  of natural numbers where  $n \geq 1$ . We call a set with  $d$  elements a  $d$ -set. Given values  $(p, k)$  for  $(\mathbf{p}, \mathbf{k})$  and a  $d$ -set  $\{c_1, \dots, c_d\}$  of intermediate computation results, we use  $D_{p,k}(c_1, \dots, c_d)$  to denote their joint distribution induced by instantiating  $\mathbf{p}$  and  $\mathbf{k}$  with  $p$  and  $k$ , respectively. Formally, for each vector of values  $v_1, \dots, v_d$  in the probability space  $\{0, 1\}^d$ , we have  $D_{p,k}(c_1, \dots, c_d)(v_1, \dots, v_d) =$

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

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

- uniformly distributed if  $D_{p,k}(c_1, \dots, c_d)$  is a uniform distribution for any  $p$  and  $k$ .
- secret independent if  $D_{p,k}(c_1, \dots, c_d) = D_{p,k'}(c_1, \dots, c_d)$  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  $\{c_1, \dots, c_k\}$  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  $D_{p,k}(c_1, \dots, c_d) \neq D_{p,k'}(c_1, \dots, c_d)$ . In this context, the main challenge is computing  $D_{p,k}(c_1, \dots, c_d)$  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 `supp` in Sect. 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 `semd` instead of `supp` as defined in Sect. 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 [6, 13, 47], together with some auxiliary data structures; then, we present our inference rules.

#### 3.1 The Type System

Let  $T = \{\text{CST}, \text{RUD}, \text{SID}, \text{NPM}, \text{UKD}\}$  be the set of distribution types for intermediate computation results, where  $\llbracket c \rrbracket$  denotes the type of  $c \leftarrow \mathbf{i}(p, k, r)$ . Specifically,

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

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

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

$$\begin{array}{c}
\text{LEAF}_1 \frac{\lambda_1(l) \in \mathbf{r}}{\llbracket l \rrbracket = \text{RUD}} \qquad \text{LEAF}_2 \frac{\lambda_1(l) \in \mathbf{p} \cup \mathbf{k}}{\llbracket l \rrbracket = \text{UKD}} \qquad \text{LEAF}_3 \frac{\lambda_1(l) \in \{0, 1\}}{\llbracket l \rrbracket = \text{CST}} \\
\text{XOR-RUD}_1 \frac{\lambda_2(l) = \oplus \quad \llbracket l.\text{lft} \rrbracket = \text{RUD} \quad \text{dom}(l.\text{lft}) \setminus \text{semd}(l.\text{rgt}) \neq \emptyset}{\llbracket l \rrbracket = \text{RUD}} \qquad \text{XOR-RUD}_2 \frac{\lambda_2(l) = \oplus \quad \llbracket l.\text{rgt} \rrbracket = \text{RUD} \quad \text{dom}(l.\text{rgt}) \setminus \text{semd}(l.\text{lft}) \neq \emptyset}{\llbracket l \rrbracket = \text{RUD}} \\
\text{AO-RUD}_1 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{rgt} \rrbracket \notin \{\text{UKD}, \text{NPM}\} \quad \text{semd}(l.\text{lft}) \cap \text{semd}(l.\text{rgt}) = \emptyset}{\llbracket l \rrbracket = \text{SID}} \qquad \text{AO-RUD}_2 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{lft} \rrbracket \notin \{\text{UKD}, \text{NPM}\} \quad \text{semd}(l.\text{rgt}) \cap \text{semd}(l.\text{lft}) = \emptyset}{\llbracket l \rrbracket = \text{SID}} \\
\text{AO-RUD}_3 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{lft} \rrbracket = \llbracket l.\text{rgt} \rrbracket = \text{RUD} \quad (\text{dom}(l.\text{lft}) \setminus \text{semd}(l.\text{rgt})) \cup (\text{dom}(l.\text{rgt}) \setminus \text{semd}(l.\text{lft})) \neq \emptyset}{\llbracket l \rrbracket = \text{SID}} \\
\text{SID} \frac{\lambda_2(l) \in \{\oplus, \wedge, \vee\} \quad \llbracket l.\text{rgt} \rrbracket = \llbracket l.\text{lft} \rrbracket = \text{SID} \quad \text{semd}(l.\text{lft}) \cap \text{semd}(l.\text{rgt}) = \emptyset}{\llbracket l \rrbracket = \text{SID}} \\
\text{NOT} \frac{\lambda_2(l) = \neg}{\llbracket l \rrbracket = \llbracket l.\text{lft} \rrbracket} \qquad \text{NO-KEY} \frac{\text{supp}(l) \cap \mathbf{k} = \emptyset}{\llbracket l \rrbracket = \text{SID}} \qquad \text{UKD} \frac{\text{no-rule is applicable at } l}{\llbracket l \rrbracket = \text{UKD}}
\end{array}$$

**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.

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

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

Figure 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  $\llbracket l \rrbracket = \text{RUD}$ . If no rule is applicable at  $l$ , we set  $\llbracket l \rrbracket = \text{UKD}$ . When the context is clear, we may use  $\llbracket l \rrbracket$  and  $\llbracket c \rrbracket$  exchangeably for  $\lambda_1(l) = c$ . The correctness of these inference rules is obvious by definition.

**Theorem 1.** *For every intermediate computation result  $c \leftarrow \mathbf{i}(\mathbf{p}, \mathbf{k}, \mathbf{r})$  labeled by  $l$ ,*

- if  $\llbracket c \rrbracket = \text{RUD}$ , then  $c$  is uniformly distributed, and hence perfectly masked;
- if  $\llbracket c \rrbracket = \text{SID}$ , then  $c$  is guaranteed to be perfectly masked.

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

*Example 1.* When using type inference with  $\text{supp}$  on the running example, we have

$$\llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket = \llbracket r_3 \rrbracket = \llbracket c_1 \rrbracket = \llbracket c_2 \rrbracket = \llbracket c_3 \rrbracket = \text{RUD}, \llbracket k \rrbracket = \llbracket c_4 \rrbracket = \llbracket c_5 \rrbracket = \llbracket c_6 \rrbracket = \text{UKD}$$

When using type inference with  $\text{semd}$  (for the second time), we have

$$\llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket = \llbracket r_3 \rrbracket = \llbracket c_1 \rrbracket = \llbracket c_2 \rrbracket = \llbracket c_3 \rrbracket = \llbracket c_4 \rrbracket = \llbracket c_5 \rrbracket = \text{RUD}, \llbracket k \rrbracket = \text{UKD}, \llbracket c_6 \rrbracket = \text{SID}$$

### 3.2 Checking Semantic Independence

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

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

where  $\Theta_s^{r=0}$  (resp.  $\Theta_s^{r=1}$ ) encodes the relation  $\mathbf{i}(\mathbf{p}, \mathbf{k}, \mathbf{r})$  with  $r$  replaced by 0 (resp. 1),  $c_0$  and  $c_1$  are copies of  $c$  and  $\Theta_s^\#$  asserts that the outputs differ even under the same inputs.

In logic synthesis and optimization, when  $r \notin \text{semd}(l)$ ,  $r$  will be called the *don't care* variable [36]. Therefore, it is easy to see why the following theorem holds.

**Theorem 2.**  $\Phi_s$  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  $\Phi_s$  is linear in  $|P|$ .

$$\begin{array}{c} \text{CP-RUD} \frac{\llbracket c_1, \dots, c_k \rrbracket = \text{RUD} \quad \llbracket c_{k+1} \rrbracket = \text{RUD} \quad \text{semd}(c_1, \dots, c_k) \cap \text{semd}(c_{k+1}) = \emptyset}{\llbracket c_1, \dots, c_{k+1} \rrbracket = \text{RUD}} \\ \text{CP-SID}_1 \frac{\llbracket c_1, \dots, c_k \rrbracket, \llbracket c_{k+1} \rrbracket \in \{\text{SID}, \text{RUD}\} \quad \llbracket c_{k+1} \rrbracket \neq \llbracket c_1, \dots, c_k \rrbracket \quad \text{semd}(c_1, \dots, c_k) \cap \text{semd}(c_{k+1}) = \emptyset}{\llbracket c_1, \dots, c_{k+1} \rrbracket = \text{SID}} \\ \text{CP-SID}_2 \frac{\llbracket c_1, \dots, c_k \rrbracket = \text{RUD} \quad \llbracket c_{k+1} \rrbracket = \text{RUD} \quad (\text{dom}(c_1, \dots, c_k) \setminus \text{semd}(c_{k+1})) \cap (\text{dom}(c_{k+1}) \setminus \text{semd}(c_1, \dots, c_k)) \neq \emptyset}{\llbracket c_1, \dots, c_{k+1} \rrbracket = \text{SID}} \\ \text{CP-UKD} \frac{\text{no-rule is applicable at } \{c_1, \dots, c_{k+1}\}}{\llbracket c_1, \dots, c_{k+1} \rrbracket = \text{UKD}} \end{array}$$

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

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

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

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



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

- if  $\llbracket c_1, \dots, c_k \rrbracket = RUD$ , then  $\{c_1, \dots, c_k\}$  is guaranteed to be uniformly distributed, and hence perfectly masked;
- if  $\llbracket c_1, \dots, c_k \rrbracket = SID$ , then  $\{c_1, \dots, c_k\}$  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  $k = 1, 2, \dots, d$ .

## 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  $c \leftarrow \mathbf{i}(\mathbf{p}, \mathbf{k}, \mathbf{r})$ , the verification of perfect masking (Definition 2) can be reduced to the *satisfiability* of the logical formula ( $\Psi$ ) defined as follows:

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

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

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

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

- $\Theta_k^r$  (resp.  $\Theta_{k'}^r$ ) for each  $r \in \{0, \dots, 2^{|\mathbf{r}|-1}\}$ : encodes a copy of the input-output relation of  $\mathbf{i}(\mathbf{p}, \mathbf{k}, r)$  (resp.  $\mathbf{i}(\mathbf{p}, \mathbf{k}', r)$ ) by replacing  $\mathbf{r}$  with concrete values  $r$ . There are  $2^{|\mathbf{r}|}$  distinct copies, but share the same plaintext  $\mathbf{p}$ .
- $\Theta_{b2i}$ : 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  $\mathbf{k}$  and  $\mathbf{k}'$ , 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:  $\llbracket c_1 \rrbracket = \llbracket c_2 \rrbracket = \llbracket c_3 \rrbracket = \llbracket c_4 \rrbracket = \llbracket c_5 \rrbracket = \llbracket c_6 \rrbracket = SID$ .

$$\begin{array}{l}
\text{AO-NPM}_1 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{rgt} \rrbracket = \text{NPM} \quad \llbracket l.\text{lft} \rrbracket = \text{RUD} \quad \text{semd}(l.\text{lft}) \cap \text{semd}(l.\text{rgt}) = \emptyset}{\llbracket l \rrbracket = \text{NPM}} \\
\text{AO-NPM}_2 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{lft} \rrbracket = \text{NPM} \quad \llbracket l.\text{rgt} \rrbracket = \text{RUD} \quad \text{semd}(l.\text{rgt}) \cap \text{semd}(l.\text{lft}) = \emptyset}{\llbracket l \rrbracket = \text{NPM}} \\
\text{AO-NPM}_3 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{rgt} \rrbracket = \text{NPM} \quad \llbracket l.\text{lft} \rrbracket = \text{RUD} \quad \text{dom}(l.\text{lft}) \setminus \text{semd}(l.\text{rgt}) \neq \emptyset}{\llbracket l \rrbracket = \text{NPM}} \\
\text{AO-NPM}_4 \frac{\lambda_2(l) \in \{\wedge, \vee\} \quad \llbracket l.\text{lft} \rrbracket = \text{NPM} \quad \llbracket l.\text{rgt} \rrbracket = \text{RUD} \quad \text{dom}(l.\text{rgt}) \setminus \text{semd}(l.\text{lft}) \neq \emptyset}{\llbracket l \rrbracket = \text{NPM}} \\
\text{CP-NPM} \frac{\llbracket c_{k+1} \rrbracket = \text{NPM}}{\llbracket c_1, \dots, c_{k+1} \rrbracket = \text{NPM}}
\end{array}$$

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

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

**Incremental SMT-Based Algorithm.** Given a computation  $c \leftarrow \mathbf{i}(p, k, r)$  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  $\text{dom}(l_s) \cap \text{unq}(l) \neq \emptyset$ . 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  $\text{dom}(l_s) \cap \text{unq}(l)$ . 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  $r \in \text{dom}(l_s) \cap \text{unq}(l)$ , which implies  $\lambda_1(l_s)$  is RUD. Here,  $r \in \text{unq}(l)$  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  $r_1 \in \text{dom}(c_2) \cap \text{unq}(c_3)$ . 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 [26, 27] can be used to simplify SMT solving. (1) When constructing formula  $\Phi$  of  $c$ , all random variables in  $\text{supp}(l) \setminus \text{semd}(l)$ , 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  $r_1 \in \text{dom}(c_2) \cap \text{unq}(c_3)$ ,  $c_2$  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 *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  $\Phi$ .

## 4.2 Feeding SMT-Based Analysis Results Back to Type System

Consider a scenario where initially the type system (cf. Sect. 3) failed to resolve a node  $l$ , i.e.,  $\llbracket l \rrbracket = \text{UKD}$ , 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  $\llbracket l \rrbracket = \text{SID}$ ; feeding it back to the type system allows us to infer more types for nodes that structurally depend on  $l$ .

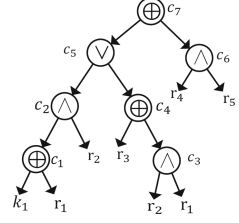


Fig. 6. Example for feeding back.

On the other hand, if SMT-based analysis shows  $l$  is not perfectly masked, the type of  $l$  can be refined to  $\llbracket l \rrbracket = \text{NPM}$ ; 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 Fig. 6. By applying the original type inference approach with either `supp` or `semd`, we have

$$\llbracket c_1 \rrbracket = \llbracket c_4 \rrbracket = \text{RUD}, \llbracket c_2 \rrbracket = \llbracket c_3 \rrbracket = \llbracket c_6 \rrbracket = \text{SID}, \llbracket c_5 \rrbracket = \llbracket c_7 \rrbracket = \text{UKD}.$$

In contrast, by applying SMT-based analysis to  $c_5$ , we can deduce  $\llbracket c_5 \rrbracket = \text{SID}$ . Feeding  $\llbracket c_5 \rrbracket = \text{SID}$  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  $\llbracket c_7 \rrbracket = \text{SID}$ . Without refinement, this was not possible.

## 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 ( $\mathbf{p}$ ), secret ( $\mathbf{k}$ ), random ( $\mathbf{r}$ ) variables and an empty map  $\pi$ , it invokes `SCINFER`( $P, \mathbf{p}, \mathbf{k}, \mathbf{r}, \pi$ ) to traverse the DDG in a topological order and annotate every node  $l$  with a distribution type from  $\mathbb{T}$ . The subroutine `TYPEINFER` implements the type inference rules outlined in Figs. 3 and 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  $f = \text{supp}$ . Once a node  $l$  is annotated as UKD, a simplified subtree  $\widehat{P}$  of the subtree rooted at  $l$  is constructed. Next, `TYPEINFER` with  $f = \text{semd}$  is invoked to resolve the UKD node in  $\widehat{P}$ . If  $\pi(l)$  becomes non-UKD afterward, `TYPEINFER` with  $f = \text{supp}$  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  $l$ .

**Algorithm 1.** Function  $\text{SCINFER}(P, p, k, r, \pi)$ 


---

```

1 Function SCINFER( $P, p, k, r, \pi$ )
2   foreach  $l \in N$  in a topological order do
3     if  $l$  is a leaf then  $\pi(l) := \llbracket l \rrbracket$ ;
4     else
5       TYPEINFER( $l, P, p, k, r, \pi, \text{supp}$ );
6       if  $\pi(l) = \text{UKD}$  then
7         let  $\widehat{P}$  be the simplified tree of the subtree rooted by  $l$  in  $P$ ;
8         TYPEINFER( $l, \widehat{P}, p, k, r, \pi, \text{semd}$ );
9         if  $\pi(l) = \text{UKD}$  then
10           res:=CheckBySMT( $\widehat{P}, p, k, r$ );
11           if res=Not-Perfectly-Masked then  $\pi(l) := \text{NPM}$ ;
12           else if res=Perfectly-Masked then  $\pi(l) := \text{SID}$ ;
13           else  $\pi(l) := \text{UKD}$ ;

```

---

**Theorem 4.** For every intermediate computation result  $c \leftarrow i(p, k, r)$  labeled by  $l$ , our method in  $\text{SCINFER}$  guarantees to return sound and complete results:

- $\pi(l) = \text{RUD}$  iff  $c$  is uniformly distributed, and hence perfectly masked;
- $\pi(l) = \text{SID}$  iff  $c$  is secret independent, i.e., perfectly masked;
- $\pi(l) = \text{NPM}$  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  $\pi(l) = \text{UKD}$ , 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.

## 5 Experiments

We have implemented our method in a verification tool named  $\text{SCINFER}$ , which uses Z3 [23] as the underlying SMT solver. We also implemented the syntactic type inference approach [47] and the incremental SMT-based approach [26,27] in the same tool for experimental comparison purposes. We conducted experiments on publicly available cryptographic software implementations, including fragments of AES and MAC-Keccak [26,27]. Our experiments were conducted on a machine with 64-bit Ubuntu 12.04 LTS, Intel Xeon(R) CPU E5-2603 v4, and 32 GB RAM.

Overall, results of our experiments show that (1)  $\text{SCINFER}$  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)  $\text{SCINFER}$  is at least twice faster than prior SMT-based verification method [26,27] on the large programs while maintaining the same accuracy; for example,  $\text{SCINFER}$  verified the benchmark named P12 in a few seconds whereas the prior SMT-based method took more than an hour.

**Algorithm 2.** Procedure  $\text{TYPEINFER}(l, P, p, k, r, \pi, f)$ 


---

```

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

```

---

## 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 [26, 27].

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

**Table 1.** Benchmark statistics.

Name	Description	#Loc	#Nodes	k	p	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

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 SC<sub>INFER</sub>, 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 SC<sub>INFER</sub>), including the number of leaky nodes (NPM type), the number of nodes actually checked by SMT, and the time.

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 SC<sub>INFER</sub> 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,

**Table 2.** Experimental results: comparison of three approaches.

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

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 `send` function, which 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.

**Table 3.** Detailed statistics of our new method.

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

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. Sect. 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 [15, 17, 34, 37, 41, 43, 46, 48, 50–52] 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 [3, 33, 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 [2,38], fault [12,29] and cache [24,35,40], or computing security bounds for probabilistic countermeasures against remote attacks [45].

Although some verification tools have been developed for this application [6,7,10,13,14,20,26,27,47], 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 [26,27]. In contrast, the model-counting based method proposed by Eldib et al. [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 [7,14,20] 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 [4,18,19,32] 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 [1,7,9,16,25,44,54] as opposed to verifying them. While methods in [1,7,9,44] rely on compiler-like pattern matching, the ones in [16,25,54] 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. Agosta, G., Barengi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 77–82 (2012)
2. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security Symposium, pp. 53–70 (2016)
3. Arribas, V., Nikova, S., Rijmen, V.: VerMI: verification tool for masked implementations. IACR Cryptology ePrint Archive, p. 1227 (2017)
4. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_15](https://doi.org/10.1007/978-3-319-21690-4_15)
5. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.-X.: On the cost of lazy engineering for masked software implementations. In: Joye, M., Moradi, A. (eds.) CARDIS 2014. LNCS, vol. 8968, pp. 64–81. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-16763-3\\_5](https://doi.org/10.1007/978-3-319-16763-3_5)
6. Barthe, G., et al.: Verified proofs of higher-order masking. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 457–485. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46800-5\\_18](https://doi.org/10.1007/978-3-662-46800-5_18)
7. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., Strub, P.-Y., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: ACM Conference on Computer and Communications Security, pp. 116–129 (2016)
8. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.* **19**(4), 769–779 (1994)
9. Bayrak, A.G., Regazzoni, F., Brisk, P., Standaert, F.-X., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 230–235 (2011)
10. Bayrak, A.G., Regazzoni, F., Novo, D., Ienne, P.: Sleuth: automated verification of software power analysis countermeasures. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 293–310. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40349-1\\_17](https://doi.org/10.1007/978-3-642-40349-1_17)
11. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Keccak implementation overview (2013). <https://keccak.team/files/Keccak-implementation-3.2.pdf>
12. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0052259>
13. Bisi, E., Melzani, F., Zaccaria, V.: Symbolic analysis of higher-order side channel countermeasures. *IEEE Trans. Comput.* **66**(6), 1099–1105 (2017)
14. Bloem, R., Gross, H., Iusupov, R., Konighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. IACR Cryptology ePrint Archive, p. 897 (2017)
15. Blömer, J., Guajardo, J., Krummel, V.: Provably secure masking of AES. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30564-4\\_5](https://doi.org/10.1007/978-3-540-30564-4_5)
16. Blot, A., Yamamoto, M., Terauchi, T.: Compositional synthesis of leakage resilient programs. In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 277–297. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54455-6\\_13](https://doi.org/10.1007/978-3-662-54455-6_13)

17. Canright, D., Batina, L.: A very compact “Perfectly Masked” S-box for AES. In: Bellare, S.M., Gennaro, R., Keromytis, A., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 446–459. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68914-0\\_27](https://doi.org/10.1007/978-3-540-68914-0_27)
18. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: AAAI Conference on Artificial Intelligence, pp. 1722–1730 (2014)
19. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 200–216. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40627-0\\_18](https://doi.org/10.1007/978-3-642-40627-0_18)
20. Coron, J.-S.: Formal verification of side-channel countermeasures via elementary circuit transformations. IACR Cryptology ePrint Archive, p. 879 (2017)
21. Coron, J.-S., Prouff, E., Rivain, M.: Side channel cryptanalysis of a higher order masking scheme. In: Paillier, P., Verbauschede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 28–44. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-74735-2\\_3](https://doi.org/10.1007/978-3-540-74735-2_3)
22. Coron, J.-S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 410–424. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43933-3\\_21](https://doi.org/10.1007/978-3-662-43933-3_21)
23. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
24. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: a tool for the static analysis of cache side channels. In: USENIX Security Symposium, pp. 431–446 (2013)
25. Eldib, H., Wang, C.: Synthesis of masking countermeasures against side channel attacks. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 114–130. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_8](https://doi.org/10.1007/978-3-319-08867-9_8)
26. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. **24**(2), 11 (2014)
27. Eldib, H., Wang, C., Schaumont, P.: SMT-based verification of software countermeasures against side-channel attacks. In: Abraham, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 62–77. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54862-8\\_5](https://doi.org/10.1007/978-3-642-54862-8_5)
28. Eldib, H., Wang, C., Taha, M., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code. In: ACM/IEEE Design Automation Conference, vol. 209, pp. 1–6 (2014)
29. Eldib, H., Wu, M., Wang, C.: Synthesis of fault-attack countermeasures for cryptographic circuits. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 343–363. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41540-6\\_19](https://doi.org/10.1007/978-3-319-41540-6_19)
30. Clavier, C., et al.: Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. J. Cryptogr. Eng. **4**(4), 259–274 (2014)
31. Fredrikson, M., Jha, S.: Satisfiability modulo counting: a new approach for analyzing privacy properties. In: ACM/IEEE Symposium on Logic in Computer Science, pp. 42:1–42:10 (2014)
32. Fremont, D.J., Rabe, M.N., Seshia, S.A.: Maximum model counting. In: AAAI Conference on Artificial Intelligence, pp. 3885–3892 (2017)
33. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side channel resistance validation. In: NIST Non-invasive Attack Testing Workshop (2011)
34. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-44709-1\\_2](https://doi.org/10.1007/3-540-44709-1_2)

35. Grabher, P., Großschädl, J., Page, D.: Cryptographic side-channels from low-power cache memory. In: Galbraith, S.D. (ed.) *Cryptography and Coding 2007*. LNCS, vol. 4887, pp. 170–184. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-77272-9\\_11](https://doi.org/10.1007/978-3-540-77272-9_11)
36. Hachtel, G.D., Somenzi, F.: *Logic Synthesis and Verification Algorithms*. Kluwer (1996)
37. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: securing hardware against probing attacks. In: Boneh, D. (ed.) *CRYPTO 2003*. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-45146-4\\_27](https://doi.org/10.1007/978-3-540-45146-4_27)
38. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) *CRYPTO 1996*. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-68697-5\\_9](https://doi.org/10.1007/3-540-68697-5_9)
39. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) *CRYPTO 1999*. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48405-1\\_25](https://doi.org/10.1007/3-540-48405-1_25)
40. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Madhusudan, P., Seshia, S.A. (eds.) *CAV 2012*. LNCS, vol. 7358, pp. 564–580. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-31424-7\\_40](https://doi.org/10.1007/978-3-642-31424-7_40)
41. Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: Goos, G., Hartmanis, J., van Leeuwen, J., Schneier, B. (eds.) *FSE 2000*. LNCS, vol. 1978, pp. 150–164. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-44706-7\\_11](https://doi.org/10.1007/3-540-44706-7_11)
42. Moradi, A., Barenghi, A., Kasper, T., Paar, C.: 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*, pp. 111–124 (2011)
43. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: a very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) *EUROCRYPT 2011*. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20465-4\\_6](https://doi.org/10.1007/978-3-642-20465-4_6)
44. Moss, A., Oswald, E., Page, D., Tunstall, M.: Compiler assisted masking. In: Prouff, E., Schaumont, P. (eds.) *CHES 2012*. LNCS, vol. 7428, pp. 58–75. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33027-8\\_4](https://doi.org/10.1007/978-3-642-33027-8_4)
45. Ochoa, M., Banescu, S., Disenfeld, C., Barthe, G., Ganesh, V.: Reasoning about probabilistic defense mechanisms against remote attacks. In: *IEEE European Symposium on Security and Privacy, EuroS&P*, pp. 499–513 (2017)
46. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A side-channel analysis resistant description of the AES S-Box. In: Gilbert, H., Handschuh, H. (eds.) *FSE 2005*. LNCS, vol. 3557, pp. 413–423. Springer, Heidelberg (2005). [https://doi.org/10.1007/11502760\\_28](https://doi.org/10.1007/11502760_28)
47. El Ouahma, I.B., Meunier, Q., Heydemann, K., Encrenaz, E.: Symbolic approach for side-channel resistance analysis of masked assembly codes. In: *Security Proofs for Embedded Systems* (2017)
48. Prouff, E., Rivain, M.: Masking against side-channel attacks: a formal security proof. In: Johansson, T., Nguyen, P.Q. (eds.) *EUROCRYPT 2013*. LNCS, vol. 7881, pp. 142–159. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38348-9\\_9](https://doi.org/10.1007/978-3-642-38348-9_9)
49. Quisquater, J.-J., Samyde, D.: ElectroMagnetic analysis (EMA): measures and counter-measures for smart cards. In: Attali, I., Jensen, T. (eds.) *E-smart 2001*. LNCS, vol. 2140, pp. 200–210. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45418-7\\_17](https://doi.org/10.1007/3-540-45418-7_17)
50. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M. (eds.) *CRYPTO 2015*. LNCS, vol. 9215, pp. 764–783. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-47989-6\\_37](https://doi.org/10.1007/978-3-662-47989-6_37)
51. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) *CHES 2010*. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-15031-9\\_28](https://doi.org/10.1007/978-3-642-15031-9_28)

52. Schramm, K., Paar, C.: Higher order masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006). [https://doi.org/10.1007/11605805\\_14](https://doi.org/10.1007/11605805_14)
53. Standaert, F.-X.: How (not) to use welch’s t-test in side-channel security evaluations. IACR Cryptology ePrint Archive 2017:138 (2017)
54. Wang, C., Schaumont, P.: Security by compilation: an automated approach to comprehensive side-channel resistance. SIGLOG News **4**(2), 76–89 (2017)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

