A Hybrid Approach to Formal Verification of Higher-Order Masked Arithmetic Programs

PENGFEI GAO, HONGYI XIE, and FU SONG, ShanghaiTech University, China
TAOLUE CHEN, University of Surrey, UK

Side-channel attacks, which are capable of breaking secrecy via side-channel information, pose a growing threat to the implementation of cryptographic algorithms. Masking is an effective countermeasure against side-channel attacks by removing the statistical dependence between secrecy and power consumption via randomization. However, designing efficient and effective masked implementations turns out to be an error-prone task. Current techniques for verifying whether masked programs are secure are limited in their applicability and accuracy, especially when they are applied. To bridge this gap, in this article, we first propose a sound type system, equipped with an efficient type inference algorithm, for verifying masked arithmetic programs against higher-order attacks. We then give novel model-counting-based and pattern-matching-based methods that are able to precisely determine whether the potential leaky observable sets detected by the type system are genuine or simply spurious. We evaluate our approach on various implementations of arithmetic cryptographic programs. The experiments confirm that our approach outperforms the state-of-the-art baselines in terms of applicability, accuracy, and efficiency.

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

Additional Key Words and Phrases: Formal verification, higher-order masking, cryptographic programs, side-channel attacks

ACM Reference format:
Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. 2021. A Hybrid Approach to Formal Verification of Higher-Order Masked Arithmetic Programs. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 26 (February 2021), 42 pages.
https://doi.org/10.1145/3428015

Pengfei Gao also with University of Chinese Academy of Sciences.
Pengfei Gao also with Shanghai Institute of Microsystem and Information Technology.
P. Gao, H. Xie, and F. Song were partially supported by the National Natural Science Foundation of China (NSFC) under Grants No.: 62072309 and No.: 61761136011. T. Chen was partially supported by the UK EPSRC Grant No.: EP/P00430X/1; the NSFC Grant No.: 61872340; the Guangdong Science and Technology Department Grant No.: 2018B010107004; the Natural Science Foundation of Guangdong Province, China No.: 2019A1515011689; and the Overseas Grant No.: KFKT2018A16 from the State Key Laboratory of Novel Software Technology, Nanjing University, China.
Authors' addresses: P. Gao, ShanghaiTech University, Shanghai, China, 201210; email: gaopf@shanghaitech.edu.cn. H. Xie, ShanghaiTech University, Shanghai, China, 201210; email: xiehy@shanghaitech.edu.cn. F. Song (corresponding author), ShanghaiTech University, Shanghai, China, 201210; email: songfu@shanghaitech.edu.cn. T. Chen, University of Surrey, Guildford, UK; email: taolue.chen@surrey.ac.uk.

1049-331X/2021/02-ART26 $15.00
https://doi.org/10.1145/3428015

1 INTRODUCTION

Cryptography as the backbone of security mechanisms plays a crucial role in many aspects of our daily lives including smart cards, cyber-physical systems, the Internet of Things, and edge computing, to name a few [2, 6, 85, 95, 99, 127, 128]. Side-channel attacks are capable of breaking secrecy via side-channel information such as power consumption [82, 89], execution time [125], faults [78, 118], acoustic [62], and cache [71], posing a growing threat to implementations of cryptographic algorithms. In this work, we focus on power side-channel attacks, where power consumption data are used as the side-channel information. Power side-channel attacks are arguably the most effective physical side-channel attack. Implementations of almost all major cryptographic algorithms, such as DES [39, 82], AES [108, 123, 128], RSA [65], Elliptic curve cryptography [41, 76, 86, 98], and post-quantum cryptography [77, 109, 112], have been successfully broken, leading to serious security implications such as cloning of GSM/3G/4G (U)SIM cards [85].

To thwart power side-channel attacks, *masking* is one of the most widely used and effective countermeasures [33, 75]. Essentially, masking is designed to remove the statistical dependence between secrecy and power consumption via randomization. Fix a sound security parameter d, and an order-d masking typically makes use of a secret-sharing scheme to logically split the secret data into $(d + 1)$ shares such that any $d' \leq d$ shares are *statistically independent* on the secret data. Masked implementations of some specific cryptographic algorithms such as PRESENT, AES, and its non-linear component (Sbox) (e.g., [75, 93, 107, 111, 114]), as well as secure conversion algorithms between Boolean and arithmetic maskings (e.g., [21, 42, 46, 64, 74]), have been published over the years. It is crucial to realize that an implementation that is based on a secure scheme does *not* provide the secure guarantee in practice automatically. For instance, the order-d masking of AES proposed in [111] and its extensions [32, 79] were later shown to be vulnerable to an attack of order-$(\lceil \frac{d}{2} \rceil + 1)$ [48]. Indeed, designing efficient and effective masked implementations is an error-prone process. Therefore, it is vital to verify masked programs in addition to the underlying security scheme, which should ideally be done automatically.

The predominant approach addressing this problem is the empirical leakage assessment by statistical significance tests or launching state-of-the-art side-channel attacks, e.g., [5, 63, 115], to cite a few. Although these approaches are able to identify some flaws, they can neither prove their absence nor identify all possible flaws exhaustively. In other words, even if no flaw is detected, it is still inconclusive, as it is entirely possible that the implementation could be broken with a better measurement setup or more leakage traces. Recently, approaches based on formal verification have been emerging for automatically verifying masked programs [8–10, 24, 25, 27, 43, 54, 55, 59, 60, 101, 129]. As the state of the art, most of these methods can only tackle Boolean programs [10, 20, 24, 25, 27, 54, 55, 129] or first-order security [59, 60, 101, 102], and thus are limited in applicability and usability. Some work [8, 9, 43] is able to verify arithmetic programs against higher-order attacks but is limited in accuracy in the sense that secure programs may fail to pass the verification, whereas potential leaky observable sets are hard to be resolved automatically, so tedious manual examination is usually necessary to differentiate genuine and spurious ones. Therefore, formal verification of masked arithmetic programs against higher-order attacks (with full tool support to automatically resolve potential leaky observable sets) is still an unsolved question and requires further research.

Main contributions. Our work focuses on formal verification of *higher-order masked arithmetic programs* based on the standard probing model (ISW model) proposed by Ishai et al. [75]. Arithmetic programs admit considerably richer operations such as finite-field multiplication and are much more challenging than their Boolean counterparts whose variables are over the Boolean domain only. Transforming arithmetic programs to equivalent Boolean ones and then applying existing tools is theoretically possible but suffers from several disadvantages: (1) complicated arithmetic operations (e.g., finite-field multiplication) have to be encoded as bitwise operations, and (2) verifying order-d security of an 8-bit arithmetic program must be done by verifying order-(8^d) security over its Boolean translation, which has considerably more observable variables (at least 8×). Because of this, we hypothesize that this approach is practically unfavorable, if not infeasible. Indeed, the state-of-the-art tool `maskVerif` [10] has already required over 18 minutes to accomplish verification of the fifth-order masked Boolean implementation of DOM Keccak Sbox [70], which has only 618 observable variables.

In light of this, we pursue a direct verification approach for higher-order masked arithmetic programs. To guarantee that a masked program is order-d secure, one has to ensure that the joint distributions of all size-d sets of observable variables (observable sets) that are potentially exposed to an attacker are independent of secret data. There are two key challenges: (1) the combinatorial explosion problem of observable sets when the number of observable variables and the security order are increasing and (2) how to efficiently and automatically resolve potential leaky observable sets. The first challenge is addressed by the **first** step of our hybrid approach, for which we propose a sound type system together with an efficient type inference algorithm, which can prescribe a *distribution type* for each observable set. One can often—but not always—deduce leakage freeness of observable sets from their distribution types, whereas observable sets that cannot be solved by the type system are regarded as potential leaky observable sets.

In case *potential* leaky observable sets are produced by the type system (i.e., the second challenge), we provide automated *resolution methods,* which are the **second** step of our hybrid approach. This step is important: for instance, [8] reported 98,176 potential third-order sets on Sbox [114], which are virtually impossible to check individually by human beings. Technically, the second step is based on *model-counting*- and *pattern-matching*-based methods. For the model counting, we consider two baseline algorithms: the first one transforms the problem to the satisfiability problem of a (quantifier-free) first-order logic formula that can be solved by SMT solvers (e.g., Z3 [50]), an extension of our previous one for first-order security [59, 60]; the second one computes the probability distribution of an observable set by naively enumerating all possible valuations of variables. We give, for the first time in the current article, a third, *GPU-accelerated parallel algorithm*, to leverage GPU's computing capability. Instead of creating a general GPU-based solver that is control-flow intensive and would downgrade the GPU performance, we automatically synthesize a GPU program for each potential leaky observable set, which, in a nutshell, enumerates all possible valuations of variables by leveraging GPU parallel computing. It turns out that the GPU-based parallel algorithm significantly outperforms the two baseline algorithms.

The pattern-matching-based method is devised to further reduce the cost of model counting. This method infers the distribution type of an observable set from observable sets whose distribution types are known, by searching an "isomorphism" between the computation expressions of the variables in two observable sets. If such an isomorphism exists, one can conclude that the two observable sets have the same distribution type, by which one can save costly model-counting procedures. The pattern-matching-based method also automatically summarizes patterns of leaky observable sets, which can be used for diagnosis and debugging.

Our hybrid approach enjoys several advantages over the existing approaches. Compared to the empirical methods based on the statistical analysis of leakage traces, our approach is able to give conclusive security assertions independent of assessment conditions, testing strategies, or the amount of gathered leakage traces. Compared to the existing formal verification approaches, our overall hybrid approach is both sound and complete, and is able to verify more types of masked implementations. Remarkably, our model-counting- and pattern-matching-based methods could also be integrated into existing formal verification approaches, effectively making them complete and more efficient.

We implement our approach in a tool HOME (**H**igher-**O**rder **M**asking v**E**rifier) and evaluate on various benchmarks, including masked implementations of full AES and MAC-Keccak programs. The results are very encouraging: HOME can handle benchmarks that have never been verified by existing formal verification approaches, e.g., implementations of Boolean to arithmetic mask conversion from [113], arithmetic to Boolean mask conversion from [45], and the non-linear transformation and round function of Simon from [117]. Our tool is also significantly faster than [8] on almost all secure programs (e.g., 110× and 31× speed-up for Key schedule [111] and fourth-order Sbox [114]; cf. Table 1), which is the only available tool to verify masked higher-order arithmetic programs under an equivalent leakage model to the ISW model. The experimental results are very encouraging in both functionality and performance when comparing our tool with the existing tools.

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

- We propose a sound type system and provide an efficient type inference algorithm for proving security of masked arithmetic programs.
- We propose a novel GPU-accelerated parallel algorithm to resolve potential leaky observable sets, which significantly outperforms two baselines.
- We propose a novel pattern-matching-based method to automatically summarize patterns of leakage sets, which can reduce the cost of model counting.
- We implement our algorithms in a software tool and demonstrate the effectiveness and efficiency of our approach on various benchmarks.

Our work can be readily used by the designers of cryptographic algorithms to verify their implementations of countermeasures against power side-channel attacks. There is, however, a potentially larger group of users. Applications from, e.g., Blockchain, the Internet of Things, edge computing, and smartphones, have used cryptographic algorithms extensively, typically provided as open source software packages that developers integrate as part of the developed software. For security-critical applications, it is vital to ensure that the entire software system is robust against power side-channel attacks where our work would play an essential role. From that perspective, average software developers—not only security experts—would potentially benefit from the current work.

The remainder of this article is organized as follows. In Section 2, we introduce basic notations and recall the probing leakage model. In Section 3, we present a motivating example and the overview of our approach. In Section 4, we present the sound type system, its inference algorithm, and sound transformations to facilitate type inference. In Section 5, we describe the model-counting- and pattern-matching-based methods. In Section 6, we evaluate the performance of our approach on representative examples from the literature. We discuss related work in Section 8. Finally, we conclude the article in Section 9.

2 PRELIMINARIES

In this section, we describe masked cryptographic programs, masking schemes, leakage models, and security notions.

2.1 Masked Cryptographic Programs

We fix an integer $\kappa > 0$ and the domain $\mathbb{I} = \{0, \ldots, 2^\kappa - 1\}$. For a set R of random variables, let $\mathcal{D}(R)$ denote the set of joint distributions over R.

Syntax. We focus on programs written in C-like code that implement cryptographic algorithms such as AES, as opposed to arbitrary software programs. The syntax is given as follows:

$$\begin{array}{lr} \text{Operation:} & \mathbf{OP} \ni \circ ::= \oplus \mid \wedge \mid \vee \mid \odot \mid + \mid - \mid \times \\ \text{Expression:} & e ::= c \in \mathbb{I} \mid x \mid e \circ e \mid \neg e \mid e \ll c \mid e \gg c \mid f(e) \\ \text{Statement:} & \texttt{stmt} ::= x \leftarrow e \mid \texttt{stmt}; \texttt{stmt} \\ \text{Program:} & P ::= \texttt{stmt};\ \texttt{return}\ x_1, \ldots, x_m; \end{array}$$

A program P is a sequence of assignments $x \leftarrow e$ followed by a `return` statement, where e is an expression building from a set of variables and κ-bit constants using the bitwise operations: *negation* ($\neg$), *and* ($\wedge$), *or* ($\vee$), *exclusive-or* ($\oplus$), *left shift* $\ll$, and *right shift* $\gg$; modulo 2^κ arithmetic operations: *addition* (+), *subtraction* (−), *multiplication* ($\times$); finite-field *multiplication* ($\odot$) over the finite field $\mathbb{F}_{2^\kappa}$; (univariate) bijective functions f, which are given by lookup tables.

To analyze a cryptographic program P, it is common to assume that it is in straight-line form (i.e., branching- and loop-free) [8, 54]. Note that our tool supports programs with non-recursive procedure calls and static loops by transforming to straight-line form by procedure inlining and loop unfolding. We currently do not include unbounded loops or recursive procedure calls. This is not a major issue as most implementations of the cryptographic algorithms feature only bounded loops and/or hierarchical procedure calls. Indeed, as one may see from Section 6, extensive and diverse benchmarks can be written in our language.

We further assume that P is in the *single static assignment* (SSA) form (i.e., there is at most one assignment $x \leftarrow e$ in P for x) and each expression uses at most one operator. (One can easily transform an arbitrary straight-line program to the SSA form.) For an assignment $x \leftarrow e$, we will denote by $\texttt{Operands}(x)$ the set of operands associated with the operator of e.

We fix a program P annotated by *public*, *private,* and *random* input variables, where the public input variables are used to store data that can be accessed by the adversary harmlessly (such as plaintext), the private input variables are used to store data that should not be accessed by the adversary (such as keys), and the random variables are sampled uniformly from the domain $\mathbb{I}$. In general, random variables are used for masking the private input variables. The set X of variables in P is partitioned into four sets: X_p, X_k, X_r, and X_i, where X_p denotes the set of public input variables, X_k denotes the set of private input variables, X_r denotes the set of (uniformly distributed) random variables on the domain $\mathbb{I}$, and X_i denotes the set of *intermediate* variables.

Semantics. For each variable $x \in X$, we define the *computation* of x, $\mathcal{E}(x)$, as an expression over input variables $X_p \cup X_k \cup X_r$. Formally, for each $x \in X$, $\mathcal{E}(x) = x$ if $x \in X_p \cup X_k \cup X_r$; otherwise, x is an intermediate variable (i.e., $x \in X_i$) that must be uniquely defined by an assignment statement $x \leftarrow e$ (thanks to SSA form of P), and thus $\mathcal{E}(x)$ is defined as the expression obtained from e by sequentially replacing all the occurrences of the intermediate variables in e by their defining expressions in P.

A *valuation* is a function $\eta : X_p \cup X_k \to \mathbb{I}$ that assigns a concrete value to each input variable in $X_p \cup X_k$. Let Θ denote the set of valuations. Two valuations η_1 and η_2 are X_p*-equivalent*, denoted by $\eta_1 \approx_{X_p} \eta_2$, if $\eta_1(x) = \eta_2(x)$ for $x \in X_p$; i.e., η_1 and η_2 must agree on their values on public input variables. We denote by $\Theta^2_{=X_p} \subseteq \Theta \times \Theta$ the set of pairs of X_p-equivalent valuations. For each variable $x \in X$, let $\mathcal{E}_\eta(x)$ denote the expression obtained from $\mathcal{E}(x)$ by instantiating variables $y \in X_p \cup X_k$ with concrete values $\eta(y)$.

Given a valuation $\eta \in \Theta$, for each variable $x \in X$, the computation $\mathcal{E}(x)$ of x under the valuation η can be interpreted as the probability distribution, denoted by $[\![x]\!]_\eta$, over the domain $\mathbb{I}$ with respect to the uniform distribution of the random variables $\mathcal{E}_\eta(x)$ may contain. Intuitively, when the values of input variables are fixed using the valuation η and the values of random variables are sampled from the domain $\mathbb{I}$, the computation $\mathcal{E}(x)$ of the variable x can be seen as a random variable with the distribution $[\![x]\!]_\eta$ defined as follows. For each concrete value $c \in \mathbb{I}$, $[\![x]\!]_\eta(c)$ is the probability that $\mathcal{E}_\eta(x)$ evaluates to c under the valuation η, that is:

$$[\![x]\!]_\eta(c) = \frac{\left|\{\mu : X_r \to \mathbb{I} \mid [\![x]\!]_{\eta,\mu} = c\}\right|}{|\mathbb{I}|^{|X_r|}},$$

where $|\{\mu : X_r \to \mathbb{I} \mid [\![x]\!]_{\eta,\mu} = c\}|$ denotes the number of assignments μ of the variables in X_r under which the computation $\mathcal{E}(x)$ evaluates to c (denoted by $[\![x]\!]_{\eta,\mu} = c$), and $|\mathbb{I}|^{|X_r|}$ denotes the number of all the possible assignments of the variables from X_r.

Accordingly, for a given valuation $\eta \in \Theta$ and a subset of variables $Y = \{x_1, \ldots, x_m\} \subseteq X$, the computations $(\mathcal{E}(x))_{x\in Y}$ under the valuation η can be interpreted as the joint distribution, denoted by $[\![P]\!]^Y_\eta$, over the domain $\mathbb{I}^m$. For each possible combination of concrete values $C = (c_1, \ldots, c_m)$ of the variables in Y, the joint distribution $[\![P]\!]^Y_\eta$ leads to the probability $[\![P]\!]^Y(C)$ that the computations $(\mathcal{E}(x))_{x\in Y}$ evaluate to $(c_1, \ldots, c_m)$ under the valuation η, that is:

$$[\![P]\!]^Y(C) = \frac{\left|\{\mu : X_r \to \mathbb{I} \mid [\![x_1]\!]_{\eta,\mu} = c_1, \ldots, [\![x_m]\!]_{\eta,\mu} = c_m\}\right|}{|\mathbb{I}|^{|X_r|}}.$$

We denote by $[\![P]\!]^Y : \Theta \to \mathcal{D}(Y)$ the function mapping of each valuation $\eta \in \Theta$ to the joint distribution $[\![P]\!]^Y_\eta$. The subscript Y may be dropped from $[\![P]\!]^Y$ and $[\![P]\!]^Y_\eta$ when $Y = X$. It is easy to see that for a given valuation $\eta \in \Theta$ and a subset of variables $Y \subseteq X$, $[\![P]\!]^Y_\eta$ is the marginal distribution of Y under the joint distribution $[\![P]\!]_\eta$ (i.e., $[\![P]\!]^X_\eta$).

2.2 Masking

Masking is a randomization technique used to break the statistical dependence of the private input variables and observable variables of the adversary [33, 75]. Fix a sound security parameter d, and an *order-d masking* typically makes use of a secret-sharing scheme to logically split the private data into $(d + 1)$ shares such that any $d' \leq d$ shares are *statistically independent* on the value of the private input. The computation of shares for each private input is usually called *presharing*. A masking transformation aims at transforming an unmasked program P that directly operates on the private inputs into a masked program P' that operates on their shares. Finally, the desired data are recovered via de-masking of the outputted shares of the masked program P'.

For example, using the order-d Boolean masking [75], the $(d + 1)$ shares of a key k are $(r_1, \ldots, r_{d+1})$, where the shares $r_1, \ldots, r_d$ are generated uniformly at random and r_{d+1} is computed such that $r_{d+1} = k \oplus \bigoplus_{i=1}^{d} r_i$. The value of k can be recovered via performing exclusive-or ($\oplus$) operations on all the shares, i.e., $\bigoplus_{i=1}^{d+1} r_i$.

Besides Boolean masking schemes, there are arithmetic masking schemes such as additive (e.g., $(k + r) \bmod n$) and multiplicative masking schemes (e.g., $(k \times r) \bmod n$) for protecting arithmetic operations. Secure conversion algorithms between them (e.g., [21, 42, 46, 64, 74]) have been proposed for masking cryptographic algorithms that embrace both Boolean and arithmetic operations (such as IDEA [84] and RC6 [40]).

When increasing the masking order d, the attack cost usually increases exponentially, but the performance of the masked programs degrades polynomially [68]. Therefore, the masking order is chosen by a trade-off between attack cost and performance.

2.3 Leakage Model and Security Notions

To formally verify the security of masked programs, it is necessary to define the set of observable variables to the adversary and a leakage model that formally captures the leaked information from the set of observable variables.

Observable variables. In the context of side-channel attacks, the adversary is assumed to be able to observe the public (X_p), random (X_r), and intermediate (X_i) variables via side-channel information, but is not able to observe the private input variables X_k or the intermediate variables of presharing. Indeed, presharing of each private input variable is performed outside of the program and is included only for verification purposes [8, 111]. Therefore, for each program P, it is easy to automatically identify the set of observable variables $X_o \subseteq X_p \cup X_i \cup X_r$ that is assumed to be observed by the adversary. Each subset $O \subseteq X_o$ of observable variables is called an *observable set.*

Leakage model. In this article, we adopt the standard d-threshold probing model proposed by Ishai et al. [75], usually referred to as the *ISW d-threshold probing model* (ISW model for short), where the adversary may have access to the values of at most d observable variables of his or her choice (e.g., via side-channel information). The more variables an adversary observes, the higher the attack cost is.

Uniform and statistical independence. Given a program P and an observable set $O \subseteq X_o$:

- P is *uniform* w.r.t. O, denoted by O*-uniform*, iff for all valuations $\eta \in \Theta$: $[\![P]\!]_\eta^O$ is a uniform joint distribution;
- P is *statistically independent* of X_k with respect to O, denoted by O-**SI**, iff for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$ (i.e., η_1 and η_2 agreeing on their values on public input variables): $[\![P]\!]_{\eta_1}^O = [\![P]\!]_{\eta_2}^O$.

We say P is O-leaky if it is *not* O-**SI**.

According to the above definitions, it is straightforward to verify the following proposition.

Proposition 2.1. *Given an observable set O of a program P,*

*(1) if P is O-uniform, then P is O-**SI** and O'-uniform for all $O' \subseteq O$;*
*(2) if P is O-**SI**, then P is O'-**SI** for all $O' \subseteq O$.*

Definition 2.2 (Security under the ISW d-threshold Probing Model [75]). A program P is *order-d secure* if P is O-**SI** or O-uniform for every observable set $O \subseteq X_o$ with $|O| = d$.

Intuitively, if P is O-**SI** or O-uniform, then the distribution of the variables in O (hence power consumptions based on the variables in O in the ISW d-threshold probing model) does not rely on private data, and thus the adversary cannot deduce any information by observing variables in O.

In the literature, the ISW d-threshold probing model is also called order-d perfect masking [54] or d-non-interference (d-NI) [8]. There are other leakage models such as the noise leakage model [107], bounded moment model [11], ISW model with transitions [44] and with glitches [25, 90], and strong d-non-interference (d-SNI) [9, 10, 58]. It is known that all these models (except for d-SNI and d-NI introduced in [9] and the extensions thereof) can be reduced to the ISW d-threshold probing model [8, 10, 11, 52] possibly at the cost of introducing higher orders when chosen plaintext attacks are adopted; namely, the adversary can use any plaintext during attack. The d-SNI and d-NI models defined in [9] are, however, stronger than the ISW d-threshold probing model. Namely, not all secure masked programs under the ISW d-threshold probing model are safe under d-SNI/d-NI and so cannot pass verification under this notion [11]. In this work, we adopt the ISW d-threshold probing model, which is more common in side-channel analysis [25, 54, 60, 129].

```
1 BooleanToArithmetic(k,r,r'){
2   x' ← k ⊕ r;  //presharing
3   y0 ← x' ⊕ r';
4   y1 ← y0 − r';
5   y2 ← y1 ⊕ x';
6   y3 ← r' ⊕ r;
7   y4 ← y3 ⊕ x';
8   y5 ← y4 − y3;
9   A ← y5 ⊕ y2;
10  return A;
11 }
```

Fig. 1. Goubin's Boolean-to-arithmetic mask conversion algorithm [64].

Remark that the d-NI notion defined in [9, 10] is strictly stronger than the one defined in [8], although they bear the same name. Namely, in [9, 10], the d-NI notion requires that the number of shares of each private input variable that can be accessed by the adversary is *strictly* less than $d + 1$.

Research objective. Our goal is to develop automated verification methods to determine whether a given masked arithmetic program is order-d secure under the ISW d-threshold probing model.

3 MOTIVATING EXAMPLE AND OVERVIEW OF APPROACHES

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

3.1 Motivating Example

Figure 1 presents an example that is an implementation of the Boolean-to-arithmetic mask conversion algorithm of Goubin [64]. The program assumes that the inputs are the private key k and two random variables r, r'. Line 2 is presharing, which computes two shares (x', r) of the private key k via Boolean masking. (Note that Line 2 should be performed outside of the function *BooleanToArithmetic* and is introduced for verification purposes only. The actual implementation in Goubin [64] takes two shares (x', r) as input and assigns r' by a uniformly sampled random value.) The function *BooleanToArithmetic* returns two shares (A, r) of the arithmetic masking of the private key k such that $A + r = k$, but without directly recovering the key k by $x' \oplus r$.

As setup for further use, we have $X_p = \emptyset, X_k = \{k\}, X_r = \{r, r'\}, X_i = \{x', A, y_1, \ldots, y_5\}$ and $X_o = \{x', A, y_0, \ldots, y_5, r, r'\}$. The computations of variables in X_i are

$$
\begin{aligned}
&\mathcal{E}(x') = k \oplus r;\\
&\mathcal{E}(y_0) = (k \oplus r) \oplus r';\\
&\mathcal{E}(y_1) = ((k \oplus r) \oplus r') - r';\\
&\mathcal{E}(y_2) = (((k \oplus r) \oplus r') - r') \oplus (k \oplus r);\\
&\mathcal{E}(y_3) = r' \oplus r;\\
&\mathcal{E}(y_4) = (r' \oplus r) \oplus (k \oplus r);\\
&\mathcal{E}(y_5) = ((r' \oplus r) \oplus (k \oplus r)) - (r' \oplus r);\\
&\mathcal{E}(A) = \Big(((r' \oplus r) \oplus (k \oplus r)) - (r' \oplus r)\Big) \oplus \Big((((k \oplus r) \oplus r') - r') \oplus (k \oplus r)\Big).
\end{aligned}
$$

For each observable variable $z \in X_o$, the program is $\{z\}$-uniform (note that $\mathcal{E}(A)$ is equivalent to $k - r$), and thus it is first-order secure. However, this program is *not* second-order secure, e.g., $y_0 \oplus y_3 \equiv x' \oplus r \equiv k$, allowing to extract private key k by observing $\{y_0, y_3\}$.

3.2 Overview of Approach

The overview of our approach HOME is depicted in Figure 2, consisting of four main components: preprocessor, type system, pattern-matching-based method, and model-counting-based method. Given a masked program P and the security order d, HOME checks whether the masked program

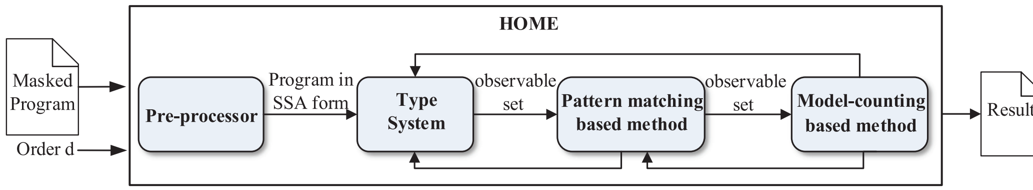

Fig. 2. Overview of our approach HOME.

P is order-d secure or not. If P is not order-d secure, then HOME outputs the leaks, i.e., all the size-d observable sets O such that P is O-leaky.

Given a masked program P and the order d, the preprocessor unfolds the static loops (i.e., loops with a predetermined bound of iterations) and inlines the procedure calls, and then transforms the program into the SSA form. The type system is used to check whether each size-d observable set O is order-d secure by deriving valid type judgements. If we can deduce that the observable set O is either order-d secure or certainly not according to the distribution type, then the result is conclusive.

However, as usual, the type system is incomplete; namely, it is possible that the distribution type cannot be inferred, in which case we first apply the pattern-matching-based method. This method iteratively searches an "isomorphism" between the computation expressions of the variables in O and the variables in O', where O' is another size-d observable set whose distribution type is already known. If such an isomorphism exists, we can conclude that these two observable sets O and O' have the same distribution type, effectively resolving the observable set O. The result of the observable set O will be fed back to the type system, which can be used to gradually improve the accuracy of the type inference.

When the pattern-matching-based method fails to resolve the observable set O, we will apply the (normally expensive) model-counting-based method, which is able to completely decide whether the observable set O is order-d secure. Finally, the observable set O is cached for further invocation of pattern matching. As before, the result of the observable set O will be fed back to the type system.

This procedure gives a sound and complete approach for verification of higher-order security. In the next two sections, we will elucidate the details of the type system, type inference algorithm, and model-counting- and pattern-matching-based methods.

4 TYPE SYSTEM

In this section, we first present a type system to infer the *distribution type* of an observable set, then propose three sound transformations to facilitate type inference, and finally present the type inference algorithm based on the type system and the sound transformations.

4.1 Dominant Variables

We first introduce the notion of dominant variables.

Definition 4.1. A random variable r is called a *dominant variable* of an expression e if the following two conditions hold:

(1) r (syntactically) occurs in the expression e exactly once; and
(2) for each operator $\circ$ on the path between the leaf r and the root in the abstract syntax tree of the expression e,
- if $\circ = \odot$, then one of its children is a non-zero constant; or
- $\circ \in \{\oplus, \neg, +, -\}$; or
- $\circ$ is a (univariate) bijective function.

Intuitively, if the computation $\mathcal{E}(x)$ contains a dominant variable r, we can immediately deduce that the distribution $[\![x]\!]_\eta$ is uniform for any valuation $\eta \in \Theta$. For instance, suppose $\mathcal{E}(x) = k \oplus r$, where k is a private input variable and r is a random variable. No matter what the value of k is, the probability $[\![x]\!]_\eta(c)$ is $\frac{1}{|\mathbb{I}|}$ for any concrete value $c \in \mathbb{I}$. Note that condition (1) in Definition 4.1 is crucial: we cannot deduce that $[\![x]\!]_\eta$ is uniform for any valuation $\eta \in \Theta$ if $\mathcal{E}(x) = (r \wedge k) \oplus r$, although it satisfies condition (2).

We denote by $\mathsf{Var}(e)$ the set of variables appearing in an expression e, by $\mathsf{RVar}(e)$ the set $\mathsf{Var}(e) \cap X_r$, and by $\mathsf{Dom}(e)$ the set of all dominant (random) variables of e. All of these sets can be computed in polynomial time in the size of e. Furthermore, note that a particularly useful example of bijective functions is Sbox, which is ubiquitous in cryptographic programs.

If r_x is a dominant variable of the expression $\mathcal{E}(x)$ such that $r_x \notin \bigcup_{x' \in O. x' \neq x} \mathsf{RVar}(\mathcal{E}(x'))$, then $\mathcal{E}(x)$ can be seen as a fresh random variable when evaluating $[\![P]\!]^O$. Therefore, if each expression $\mathcal{E}(x)$ for $x \in O$ has such dominant variables, we can deduce that P is O-uniform.

Proposition 4.2. *Given an observable set $O \subseteq X_o$, if for every $x \in O$ there exists a dominant variable $r_x \in \mathsf{Dom}(\mathcal{E}(x))$ such that $r_x \notin \bigcup_{x' \in O. x' \neq x} \mathsf{RVar}(\mathcal{E}(x'))$, then P is O-uniform.*

Proof. To prove this proposition, we first introduce the notion of i-invertibility. We will denote by $e(x_1, \ldots, x_n)$ an expression defined over the variables $x_1, \ldots, x_n$, which can be seen as a function mapping a combination of concrete values $c_1, \ldots, c_n$ to a concrete value by instantiating all the variables $x_1, \ldots, x_n$ with their corresponding concrete values $c_1, \ldots, c_n$. An expression $e(x_1, \ldots, x_n)$ is *i-invertible* if, for any concrete values $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n \in \mathbb{I}$, the expression

$$e(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$$

obtained by instantiating all the variables $(x_j)_{j \neq i}$ with concrete values $(c_j)_{j \neq i}$ is bijective. It is easy to see that $e(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$ and x_i have same distribution. Thus, if x_i is a random variable, then $e(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$ must be a uniform distribution.

The following claim reveals the relation between dominated expressions and i-invertibility.

Claim. Given an expression $e(x_1, \ldots, x_n)$ over variables $\{x_1, \ldots, x_n\}$, for every $1 \leq i \leq n$, if x_i is a dominant variable of $e(x_1, \ldots, x_n)$, then $e(x_1, \ldots, x_n)$ is i-invertible.

We prove that $e(x_1, \ldots, x_n)$ is i-invertible by induction on the length ℓ of the path between the leaf x_i and the root in the abstract syntax tree of e.

Base case $\ell = 0$. The expression $e(x_1, \ldots, x_n)$ must be x_i, which is a bijective function. The result immediately follows.

Inductive step $\ell > 0$. Let $\circ$ be the operator at the root of the syntax tree of $e(x_1, \ldots, x_n)$; then $e(x_1, \ldots, x_n)$ is in the form of

(1) $\neg e_1(x_1, \ldots, x_n)$; or
(2) $\circ(e_1(x_1, \ldots, x_n))$, where $\circ$ is a (univariate) bijective function; or
(3) $e_1(x_1, \ldots, x_n) \circ e_2(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ such that x_i is a dominant variable of $e_1(x_1, \ldots, x_n)$, where $\circ \in \{\odot, \oplus, +, -\}$. (Note that x_i does not appear in $e_2(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$.)

By the induction hypothesis, $e_1(x_1, \ldots, x_n)$ is i-invertible. By the definition of i-invertibility, for any concrete values $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n \in \mathbb{I}$, $e_1(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$

is bijective. Then, the result immediately follows if $e(x_1, \ldots, x_n)$ is $\neg e_1(x_1, \ldots, x_n)$ or $\circ(e_1(x_1, \ldots, x_n))$, i.e., Item (1) and Item (2). It remains to consider Item (3).

- If $\circ = \odot$, then $e_2(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ is a non-zero constant. Note that x_i is a dominant variable of $e_1(x_1, \ldots, x_n)$. By applying the induction hypothesis, $e_1(x_1, \ldots, x_n)$ is i-invertible; therefore, $e_1(x_1, \ldots, x_n)$ cannot be a constant. Since the multiplicative group of the non-zero elements in $\mathbb{I}$ is cyclic and $0 \odot e_2(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = 0$, then $e_1(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n) \odot e_2(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$ is also bijective. Hence, the result follows.
- If $\circ \in \{\oplus, +, -\}$, then $e_2(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, c_{i+1}/x_{i+1}, \ldots, c_n/x_n)$ is a constant. For any constant $c \in \mathbb{I}$, $e_1(c_1/x_1, \ldots, c_{i-1}/x_{i-1}, x_i, c_{i+1}/x_{i+1}, \ldots, c_n/x_n) \circ c$ is still bijective (note that $+$ and $-$ are operators over the ring $\mathbb{I}$). Hence, the result follows.

Now, we prove the proposition.

Suppose for every $x \in O$, there exists $r_x \in \mathsf{Dom}(\mathcal{E}(x))$ such that $r_x \notin \bigcup_{x' \in O. x' \neq x} \mathsf{RVar}(\mathcal{E}(x'))$; let $[\![P[r_x/x]_{x \in O}]\!]_\eta^O$ denote the distribution of $[\![P]\!]_\eta^O$ in which $\mathcal{E}(x)$ is replaced by r_x for all $x \in O$; then for all valuations $\eta \in \Theta$, $[\![P]\!]_\eta^O = [\![P[r_x/x]_{x \in O}]\!]_\eta^O$ holds.

By applying the above claim, we get that $[\![P[r_x/x]_{x \in O}]\!]_\eta^O$ is a uniform distribution. Therefore, the result immediately follows. □

Note that the condition $r_x \notin \bigcup_{x' \in O. x' \neq x} \mathsf{RVar}(\mathcal{E}(x'))$ is crucial in Proposition 4.2, because otherwise the distributions of some variables may be statistically dependent. For instance, consider the observable set $O = \{x, y\}$ with $\mathcal{E}(x) = k \oplus r$ and $\mathcal{E}(y) = r$, where k is a private input variable and r is a random variable. The program is both $\{x\}$-uniform and $\{y\}$-uniform but is not O-uniform.

Example 4.3. Let us consider the motivating example in Section 3.1. $\mathcal{E}(x')$ is dominated by the random variable r. $\mathcal{E}(y_0)$ and $\mathcal{E}(y_3)$ both have two dominant variables r and r'. $\mathcal{E}(y_1)$ only has the dominant variable r, as r' occurs twice. Similarly, $\mathcal{E}(y_4)$ only has the dominant variable r', as r occurs twice. Thus, for every observable set $O \subseteq \{x', y_0, y_1, y_3, y_4\}$ with $|O| = 1$, we can deduce that the program is O-uniform. $\mathcal{E}(y_2)$, $\mathcal{E}(y_5)$, and $\mathcal{E}(A)$ have no dominant variables, as both r and r' occur more than once.

For the observable set $\{x', y_3\}$, although the dominant variable r' of $\mathcal{E}(y_3)$ does not appear in $\mathcal{E}(x')$, the dominant variable r of $\mathcal{E}(x')$ appears in $\mathcal{E}(y_3)$, and thus we cannot deduce that the program is $\{x', y_3\}$-uniform. Indeed, for any observable set $O \subseteq \{x', A, y_1, \ldots, y_5, r, r'\}$ with $|O| \geq 2$, we cannot deduce that the program is O-uniform.

4.2 Types and Type Inference Rules

In this subsection, we introduce distribution types and their inference rules for proving higher-order security.

Definition 4.4. Let $\mathcal{T}$ be the set of (distribution) types $\{\tau_{\mathtt{uf}}, \tau_{\mathtt{si}}, \tau_{\mathtt{lk}}\}$:

- $\tau_{\mathtt{uf}}$ stands for uniform distribution; i.e., $O : \tau_{\mathtt{uf}}$ means that the program is O-uniform;
- $\tau_{\mathtt{si}}$ stands for secret independent distribution; i.e., $O : \tau_{\mathtt{si}}$ means that the program is O-**SI**;
- $\tau_{\mathtt{lk}}$ stands for leak; i.e., $O : \tau_{\mathtt{lk}}$ means that the program is O-leaky, namely, not O-**SI**,

where O is an observable set.

The distribution type $\tau_{\mathtt{uf}}$ is a subtype of $\tau_{\mathtt{si}}$; i.e., $\tau_{\mathtt{uf}}$ implies $\tau_{\mathtt{si}}$, but $\tau_{\mathtt{si}}$ does not imply $\tau_{\mathtt{uf}}$. Although both $\tau_{\mathtt{si}}$ and $\tau_{\mathtt{uf}}$ can be used to prove that the program is statistically independent of the secret for an observable set O, i.e., no leak, $\tau_{\mathtt{uf}}$ is more desired because the observable set O

$$\frac{\vdash x_1 \star x_2 : \tau \qquad x \leftarrow x_2 \star x_1}{\vdash x : \tau}\ (\textsc{Com}) \qquad \frac{\vdash x' : \tau \qquad x \leftarrow \neg x'}{\vdash x : \tau}\ (\textsc{Ide}_1)$$

$$\frac{\vdash x' : \tau_{\mathtt{si}} \qquad x \leftarrow x' \bullet x'}{\vdash x : \tau_{\mathtt{si}}}\ (\textsc{Ide}_2) \qquad \frac{x \leftarrow x' \diamond x'}{\vdash x : \tau_{\mathtt{si}}}\ (\textsc{Ide}_3)$$

$$\frac{\vdash x' : \tau_{\mathtt{lk}} \qquad x \leftarrow x' \bowtie x'}{\vdash x : \tau_{\mathtt{lk}}}\ (\textsc{Ide}_4) \qquad \frac{\vdash x_1 : \tau_{\mathtt{uf}} \qquad \vdash x_2 : \tau_{\mathtt{uf}} \qquad x \leftarrow x_1 \circ x_2 \qquad \mathsf{Dom}(\mathcal{E}(x_1)) \setminus \mathsf{RVar}(\mathcal{E}(x_2)) \neq \emptyset}{\vdash x : \tau_{\mathtt{si}}}\ (\textsc{Sid}_4)$$

$$\frac{\vdash x_1 : \tau_{\mathtt{si}} \qquad \vdash x_2 : \tau_{\mathtt{si}} \qquad x \leftarrow x_1 \bullet x_2 \qquad \mathsf{RVar}(\mathcal{E}(e_1)) \cap \mathsf{RVar}(\mathcal{E}(e_2)) = \emptyset}{\vdash x : \tau_{\mathtt{si}}}\ (\textsc{Sid}_5) \qquad \frac{\vdash x_1 : \tau_{\mathtt{lk}} \qquad \vdash x_2 : \tau_{\mathtt{uf}} \qquad x \leftarrow x_1 \circ x_2 \qquad \mathsf{Dom}(\mathcal{E}(x_2)) \setminus \mathsf{RVar}(\mathcal{E}(x_1)) \neq \emptyset}{\vdash x : \tau_{\mathtt{lk}}}\ (\textsc{Sdd})$$

Fig. 3. Type inference rules for first-order security, where $\star \in \mathbf{OP}$, $\circ \in \{\wedge, \vee, \odot, \times\}$, $\bullet \in \mathbf{OP}^*$, $\bowtie \in \{\wedge, \vee\}$, and $\diamond \in \{\oplus, -\}$.

$$\frac{(\bigcup_{x \in O} \mathsf{Var}(\mathcal{E}(x))) \cap X_k = \emptyset}{\vdash O : \tau_{\mathtt{si}}}\ (\textsc{No-Key}) \qquad \frac{\vdash O_1 : \tau_{\mathtt{si}} \qquad \bigcup_{x \in O_2} \mathsf{Var}(\mathcal{E}(x)) \subseteq X_p}{\vdash O_1 \cup O_2 : \tau_{\mathtt{si}}}\ (\textsc{Sid}_1)$$

$$\frac{\vdash O : \tau_{\mathtt{si}} \qquad \mathtt{Operands}(x) \subseteq O \cup X_p \cup \mathbb{I}}{\vdash O \cup \{x\} : \tau_{\mathtt{si}}}\ (\textsc{Sid}_2)$$

$$\frac{\vdash O_1 : \tau_{\mathtt{si}} \qquad \forall x \in O_2. \exists r_x \in \mathsf{Dom}(\mathcal{E}(x)) \setminus \bigcup_{y \in O_1 \cup O_2 \wedge y \neq x} \mathsf{RVar}(\mathcal{E}(y))}{\vdash O_1 \cup O_2 : \tau_{\mathtt{si}}}\ (\textsc{Sid}_3)$$

$$\frac{\vdash O_1 : \tau_{\mathtt{uf}} \qquad \forall x \in O_2. \exists r_x \in \mathsf{Dom}(\mathcal{E}(x)) \setminus \bigcup_{y \in O_1 \cup O_2 \wedge y \neq x} \mathsf{RVar}(\mathcal{E}(y))}{\vdash O_1 \cup O_2 : \tau_{\mathtt{uf}}}\ (\textsc{Rud})$$

Fig. 4. Type inference rules for **higher-order** security.

not only is statistically independent of the secret (same as in $\tau_{\mathtt{si}}$) but also can be used like a set of random variables. Therefore, we prefer $\tau_{\mathtt{uf}}$ over $\tau_{\mathtt{si}}$ and want to infer as many $\tau_{\mathtt{uf}}$ as possible.

Type judgments are in the form of $\vdash O : \tau$, where O is an observable set, and $\tau \in \mathcal{T}$ is the type of O. Note that we omitted the context of the type judgment for simplifying presentation. The type judgment $\vdash O : \tau$ is valid iff the distribution of the values of variables from O satisfies the property specified by τ in the program P.

Figure 3 presents type inference rules for the *first-order* security. We denote by $\mathbf{OP}^*$ the set $\mathbf{OP} \cup \{\ll, \gg\}$. Rule (Com) captures the commutative law of operators $\star \in \mathbf{OP}$. Rules (Ide$_i$) for $i = 1, 2, 3, 4$ are straightforward. Rule (Sid$_4$) states that x has type $\tau_{\mathtt{si}}$ if $x \leftarrow x_1 \circ x_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$, both x_1 and x_2 have type $\tau_{\mathtt{uf}}$, and $\mathcal{E}(x_1)$ has a dominant variable r that is not used by $\mathcal{E}(x_2)$. Indeed, $\mathcal{E}(x)$ can be seen as $r \circ \mathcal{E}(x_2)$. Rule (Sid$_5$) states that expression x has type $\tau_{\mathtt{si}}$ if $x \leftarrow x_1 \bullet x_2$ for $\bullet \in \mathbf{OP}^*$, both x_1 and x_2 have type $\tau_{\mathtt{si}}$ (as well as its subtype $\tau_{\mathtt{uf}}$), and the sets of random variables used by $\mathcal{E}(x_1)$ and $\mathcal{E}(x_2)$ are disjoint. Indeed, for each valuation $\eta \in \Theta$, the distributions $[\![x_1]\!]_\eta$ and $[\![x_2]\!]_\eta$ are independent. Rule (Sdd) states that the variable x has type $\tau_{\mathtt{lk}}$ if $x \leftarrow x_1 \circ x_2$ for $\circ \in \{\wedge, \vee, \odot, \times\}$, x_1 has type $\tau_{\mathtt{lk}}$, x_2 has type $\tau_{\mathtt{uf}}$, and $\mathcal{E}(x_2)$ has a dominant variable r that is not used by $\mathcal{E}(x_1)$. Intuitively, $\mathcal{E}(x)$ can be safely seen as $\mathcal{E}(x_1) \circ r$.

Note that the type inference rules for the first-order security are similar to those from [60], which are reproduced here for completeness. The new rules for the higher-order security are given in Figure 4. We briefly explain these rules below.

Rule (No-Key) states that if O is an observable set whose values are independent of private variables, then O has type $\tau_{\mathtt{si}}$. Rule (Sid$_1$) states that if O_1 has type $\tau_{\mathtt{si}}$ and the computations $\mathcal{E}(x)$ of variables $x \in O_2$ only involve public variables, then we can deduce that $O_1 \cup O_2$ has type $\tau_{\mathtt{si}}$.

Rule (SID$_2$) states that if O has type $\tau_{\mathtt{si}}$ and a variable x is defined using constants, public variables, or variables in O, then adding x into O does not change the type. Intuitively, as the value of x is determined by its operands, for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, $[\![P]\!]^O_{\eta_1} = [\![P]\!]^O_{\eta_2}$ if and only if $[\![P]\!]^{O\cup\{x\}}_{\eta_1} = [\![P]\!]^{O\cup\{x\}}_{\eta_2}$. Rule (SID$_3$) deals with a $\tau_{\mathtt{si}}$-typed observable set O_1 and a $\tau_{\mathtt{uf}}$-typed observable set O_2 (cf. Proposition 4.2). Assume that each computation $\mathcal{E}(x)$ for $x \in O_2$ has a dominant variable r_x that is not used in any computation of variable in $O_1 \cup O_2$ except x; then $O_1 \cup O_2$ has type $\tau_{\mathtt{si}}$. Intuitively, each computation $\mathcal{E}(x)$ for $x \in O_2$ can be seen as the random variable r_x, and O_1 has type $\tau_{\mathtt{si}}$; hence, the distributions $[\![x]\!]_\eta$ for all x in $O_1 \cup O_2$ and all valuations $\eta \in \Theta$ are independent. Similarly, rule (RUD) deals with two $\tau_{\mathtt{uf}}$-typed observable sets O_1 and O_2. Although the dominant variables of $\mathcal{E}(x)$ for $x \in O_1$ may appear in the computations of variables in O_2, by Proposition 4.2, all the variables $x \in O_2$ can be seen as fresh random variables r_x so that Proposition 4.2 can be applied.

Note that the type $\tau_{\mathtt{lk}}$ can only be derived in the inference rules for the first-order security. We could introduce another inference rule for the higher-order security that asserts that an observable set O has the type $\tau_{\mathtt{lk}}$ if any subset of O has type $\tau_{\mathtt{lk}}$. We do not present this inference rule in this work as it will not be used in our type inference algorithm.

THEOREM 4.5 (SOUNDNESS OF THE TYPE SYSTEM). *For every set observable $O \subseteq X_o$,*

*(1) if $\vdash O : \tau_{\mathtt{si}}$ is valid, then P is O-*SI*;*
(2) if $\vdash O : \tau_{\mathtt{uf}}$ is valid, then P is O-uniform;
(3) if $\vdash O : \tau_{\mathtt{lk}}$ is valid, then P is O-leaky.

PROOF. We only show the soundness of rules for the higher-order security. First, Item (3) directly follows from the first-order case [60]. We now deal with the rules in Figure 4.

- **Rule** (NO-KEY). Suppose $(\bigcup_{x\in O} \mathsf{Var}(\mathcal{E}(x))) \cap X_k = \emptyset$; then the expression $\mathcal{E}(x)$ does not use any private variable for all $x \in O$. This implies that $[\![P]\!]^O_{\eta_1} = [\![P]\!]^O_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$ (note that η_1 and η_2 must agree on their values on public input variables).
- **Rule** (SID$_1$). Suppose $\vdash O_1 : \tau_{\mathtt{si}}$; then $[\![P]\!]^{O_1}_{\eta_1} = [\![P]\!]^{O_1}_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$. Consider O_2 such that $\bigcup_{x\in O_2} \mathsf{Var}(\mathcal{E}(x)) \subseteq X_p$; then for every $x \in O_2$, $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, and assignment of random variables $f : X_r \to \mathbb{I}$, the expression $\mathcal{E}(x)$ evaluates to same value under (η_1, f) and (η_2, f). This implies that $[\![P]\!]^{O_1\cup O_2}_{\eta_1} = [\![P]\!]^{O_1\cup O_2}_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$.
- **Rule** (SID$_2$). Suppose $\vdash O : \tau_{\mathtt{si}}$; then $[\![P]\!]^O_{\eta_1} = [\![P]\!]^O_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$. Suppose the observable set O is $\{x_1, \dots, x_n\}$; then for each vector of concrete values $(c_1, \dots, c_n) \in \mathbb{I}^n$, $[\![P]\!]^O_{\eta_1}(c_1, \dots, c_n) = [\![P]\!]^O_{\eta_2}(c_1, \dots, c_n)$. Consider x_{n+1} such that $\mathtt{Operands}(x_{n+1}) \subseteq O \cup X_p \cup \mathbb{I}$; let c_{n+1} denote the value of x_{n+1} under the valuation η_1 and $x_1 = c_1, \dots, x_n = c_n$, and c'_{n+1} denote the value of x_{n+1} under the valuation η_2 and $x_1 = c_1, \dots, x_n = c_n$. Since η_1 and η_2 must agree on their values on public input variables, then $c_{n+1} = c'_{n+1}$. Therefore, for every concrete value c, $[\![P]\!]^{O\cup\{x_{n+1}\}}_{\eta_1}(c_1, \dots, c_n, c) = [\![P]\!]^{O\cup\{x_{n+1}\}}_{\eta_2}(c_1, \dots, c_n, c) = 0$ if $c \neq c_{n+1}$, $[\![P]\!]^{O\cup\{x_{n+1}\}}_{\eta_1}(c_1, \dots, c_n, c) = [\![P]\!]^O_{\eta_1}(c_1, \dots, c_n)$ and $[\![P]\!]^{O\cup\{x_{n+1}\}}_{\eta_2}(c_1, \dots, c_n, c) = [\![P]\!]^O_{\eta_2}(c_1, \dots, c_n)$ if $c = c_{n+1}$. Hence, the result immediately follows.
- **Rule** (SID$_3$). Suppose $\vdash O_1 : \tau_{\mathtt{si}}$; then $[\![P]\!]^{O_1}_{\eta_1} = [\![P]\!]^{O_1}_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$. Consider O_2 such that $\forall x \in O_2.\exists r_x \in \mathsf{Dom}(\mathcal{E}(x)) \setminus \bigcup_{y\in O_1\cup O_2 \wedge x\neq y} \mathsf{RVar}(\mathcal{E}(y))$; i.e., for each $x \in O_2$, there exists a dominant random variable $r_x \in \mathsf{Dom}(\mathcal{E}(x))$ that is not used in other expressions in $\mathcal{E}(y)$ for $y \in O_1 \cup O_2$ with $x \neq y$. Thus, $[\![P]\!]^{O_1\cup O_2}_{\eta_1} = [\![P]\!]^{O_1\cup O_2}_{\eta_2}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$.

- **Rule** (RUD). Consider $\mathcal{O}_2$ such that $\forall x \in \mathcal{O}_2.\exists r_x \in \mathsf{Dom}(\mathcal{E}(x)) \setminus \bigcup_{y\in\mathcal{O}_1\cup\mathcal{O}_2\wedge x\neq y} \mathsf{RVar}(\mathcal{E}(y))$; i.e., for each $x \in \mathcal{O}_2$, there exists a dominant random variable $r_x \in \mathsf{Dom}(\mathcal{E}(x))$ that is not used in other expressions in $\mathcal{E}(y)$ for $y \in \mathcal{O}_1 \cup \mathcal{O}_2$ with $x \neq y$. Let $\mathcal{O}_3$ denote the set of such random variables r_x. Then $[\![P]\!]_{\eta_1}^{\mathcal{O}_1\cup\mathcal{O}_2} = [\![P]\!]_{\eta_2}^{\mathcal{O}_1\cup\mathcal{O}_3}$ for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$.

  Since the program is $\mathcal{O}_1$-uniform, then we get that the program is $\mathcal{O}_1 \cup \mathcal{O}_3$-uniform. The result follows from $[\![P]\!]_{\eta_1}^{\mathcal{O}_1\cup\mathcal{O}_2} = [\![P]\!]_{\eta_2}^{\mathcal{O}_1\cup\mathcal{O}_3}$.

The proof is completed. □

Note that our type inference rules are designed to be redundant for efficiency consideration. Namely, they have distinct complexities to check the premises. For instance, rule (RUD) is a special case of rule (SID$_3$), as τ_{uf} is a subtype of τ_{si}, but we prefer τ_{uf} over τ_{si}. Also, rule (RUD) is a reformation and generalization of Proposition 4.2, which allows to add more observable variables to the observable set $\mathcal{O}_1$ without searching dominant variables in the computations of variables in $\mathcal{O}_1$. A valid judgment derived by rule (SID$_2$) in constant time can also be derived by using other rules, but rule (SID$_2$) could avoid unfolding the definitions of variables. When applying these rules, we start with those which can infer the type τ_{uf} and whose premises can be established at a lower cost, namely, in the order of rules (RUD), (SID$_2$), (SID$_1$), (NO-KEY), and (SID$_3$).

Example 4.6. Let us consider the motivating example in Section 3.1. Recalling that $\mathcal{E}(y_3) = r' \oplus r$, although we can derive both $\vdash \{y_3\} : \tau_{\mathsf{uf}}$ by applying rule (RUD) and $\vdash \{y_3\} : \tau_{\mathsf{si}}$ by applying rule (NO-KEY), we will prefer $\vdash \{y_3\} : \tau_{\mathsf{uf}}$.

In Example 4.3, we claim that for any observable set $\mathcal{O} \subseteq \{x', A, y_1, \ldots, y_5, r, r'\}$ with $|\mathcal{O}| \geq 2$, we cannot deduce that the program is $\mathcal{O}$-uniform by applying Proposition 4.2. As an example, let us consider the observable set $\{x', y_3\}$. As $\mathcal{E}(x')$ is dominated by the random variable r, we derive that $\vdash \{x'\} : \tau_{\mathsf{uf}}$. As $\mathcal{E}(y_3)$ has the dominant variable r' that does not appear in $\mathcal{E}(x')$, we can derive that $\vdash \{x', y_3\} : \tau_{\mathsf{uf}}$ by applying rule (RUD). Indeed, since the dominant variable r' of $\mathcal{E}(y_3)$ does not appear in $\mathcal{E}(x')$, we can safely regard $\mathcal{E}(y_3)$ as the dominant variable r' so that the dominant variable r of $\mathcal{E}(x')$ is eliminated from $\mathcal{E}(y_3)$. This allows to apply Proposition 4.2 to prove that the program is $\{x', y_3\}$-uniform. However, if we do not regard $\mathcal{E}(y_3)$ as the dominant variable r', then r appears in $\mathcal{E}(y_3)$, so we cannot directly apply Proposition 4.2 to prove that the program is $\{x', y_3\}$-uniform. Similarly, we can deduce that $\vdash \{x', y_0\} : \tau_{\mathsf{uf}}$ and $\vdash \{x', y_4\} : \tau_{\mathsf{uf}}$, but we still cannot deduce the distribution types of the other size-2 observable sets. For instance, we cannot infer the distribution type of the observable set $\{x', y_1\}$, as $\mathsf{Dom}(\mathcal{E}(x')) = \mathsf{Dom}(\mathcal{E}(y_1)) = \{r\}$.

4.3 Sound Transformations

In this subsection, we describe three sound, domain-specific transformations for facilitating type inference.

The first transformation is based on the observation that some computations may share common sub-expressions that are dominated by some random variables, and these random variables are only used in these sub-expressions. Such sub-expressions, treated as random variables (i.e., replaced by the dominant variables) when analyzing the computations, are uniform and independent. This may enable type inference rules, as the other random variables in sub-expressions will be eliminated. Therefore, we can simplify computations by leveraging the notion of dominant variables.

For instance, consider the observable set $\{x', y_1\}$ in the motivating example. Recall that $\mathcal{E}(x') = k \oplus r$ and $\mathcal{E}(y_1) = ((k \oplus r) \oplus r') - r'$. We can observe that the sub-expression $k \oplus r$ is dominated by the random variable r, which occurs exclusively in $k \oplus r$. Therefore, $\mathcal{E}(x')$ and $\mathcal{E}(y_1)$ can be simplified as r and $(r \oplus r') - r'$, respectively, as the distributions of $v \oplus r$ and r are identical for any value $v \in \mathbb{I}$ of k, and r does not affect the values of other sub-expressions. Using the simplified computations r and $(r \oplus r') - r'$ of $\mathcal{E}(x')$ and $\mathcal{E}(y_1)$, we can deduce $\vdash \{x', y_1\} : \tau_{\mathtt{si}}$ by applying rule (No-Key). This simple, but crucial, observation is formalized as the following definition.

Definition 4.7. A sub-expression e in a set of computations E is *dominated by a random variable* r if $r \in \mathsf{Dom}(e)$ and r only occurs in e, namely, does not occur in E elsewhere.

To facilitate type inference, we may replace the largest r-dominated sub-expression e by r, which can be done in polynomial time by traversing the abstract syntax tree. Let $\mathsf{Simply}_{\mathsf{Dom}}(E)$ be the set of computations obtained from E by repeatedly applying this strategy. We remark that $\mathsf{Simply}_{\mathsf{Dom}}(E)$ is more general than Proposition 4.2. If an observable set $O \subseteq X_o$ satisfies the conditions in Proposition 4.2, i.e., for every $x \in O$ there exists a dominant variable $r_x \in \mathsf{Dom}(\mathcal{E}(x))$ such that $r_x \notin \bigcup_{x' \in O. x' \neq x} \mathsf{RVar}(\mathcal{E}(x'))$, then the computations $(\mathcal{E}(x))_{x \in O}$ can be replaced by the random variables $(r_x)_{x \in O}$ from which we can directly deduce that P is O-uniform. However, $\mathsf{Simply}_{\mathsf{Dom}}(E)$ is also computationally more expensive than checking the conditions in Proposition 4.2. Therefore, we apply $\mathsf{Simply}_{\mathsf{Dom}}(E)$ only when the type system fails.

$\mathsf{Simply}_{\mathsf{Dom}}$ is generally very effective in our experiments but fails on one benchmark. This is because $\mathsf{Simply}_{\mathsf{Dom}}$ only relies upon syntactic information of the computation. For instance, consider the observable set $\{x_1, x_2\}$ taking from the second-order masked implementation of the AES Sbox [114], where

- $\mathcal{E}(x_1) = \Big(\mathsf{Sbox}\big((0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus r_2\Big) \oplus r_3,$
- $\mathcal{E}(x_2) = \Big(\mathsf{Sbox}\big((r_0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus r_2\Big) \oplus r_3,$

k is a private input variable, and r_0, r_1, r_2, r_3 are random variables. $\mathsf{Simply}_{\mathsf{Dom}}$ is not able to simplify the sub-expression $r_2 \oplus r_3$ into a random variable, though both r_2 and r_3 are dominant variables of $r_2 \oplus r_3$.

$\mathsf{Simply}_{\mathsf{Dom}}$ could be applied if we could transform $\mathcal{E}(x_1)$ and $\mathcal{E}(x_2)$ to equivalent forms (by the associativity of $\oplus$), i.e.,

$$\mathsf{Sbox}\big((0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus (r_2 \oplus r_3) \text{and} \mathsf{Sbox}\big((r_0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus (r_2 \oplus r_3).$$

However, carrying out such a transformation automatically is very challenge in general, as there is no canonical representation of the computation to which $\mathsf{Simply}_{\mathsf{Dom}}$ can be applied. To address this challenge, we propose the sound transformation that aims to collapse several variables into one variable, e.g., collapse r_2 and r_3 into a new random variable even if they do not appear as the sub-expression $r_2 \oplus r_3$. This idea is formalized as the following definition.

Definition 4.8. Given a set of computations E and a set of variables $Z \subseteq \bigcup_{e \in E} \mathsf{Var}(e)$, Z is *collapsible* with respect to E if the following two conditions hold:

(1) $Z \subseteq X_p$ or $Z \subseteq X_k$ or $Z \subseteq X_r$; namely, variables in Z have the same type; and
(2) there exist sub-expressions $e_1, \ldots, e_m$ in E such that:
  - sub-expression e_j for each $1 \le j \le m$ can be rewritten as $(\bigoplus_{z \in Z} z) \oplus e'_j$, i.e., clustering the variables in Z together, and
  - each variable $z \in Z$ only occurs in $\{e_1, \ldots, e_m\}$ and occurs in e_j for each $1 \le j \le m$ exactly once.

One can observe that if Z is collapsible, then $\bigoplus_{z \in Z} z$ can be replaced by a fresh variable respecting the type (i.e., public, key, or random) when analyzing $\{\mathcal{E}(x) \mid x \in O\}$ for the observable set O. For simplicity, we usually use $\overline{Z}$ to denote the fresh variable. We denote by $\mathsf{Simply}_{\mathsf{Col}}(E)$

the set of computations computed from E by repeatedly applying this strategy. $\mathsf{Simply}_{\mathsf{Col}}(E)$ is implemented in polynomial time by iteratively searching pairs of variables $\{x_1, x_2\}$ that are collapsible and replacing them by $\overline{\{x_1, x_2\}}$.

The third transformation is the application of algebra laws. We denote by $\mathsf{Simply}_{\mathsf{Alg}}(E)$ the set of computations computed from E by repeatedly applying algebra laws such as $e \oplus e \equiv 0$, $0 \oplus e \equiv e$, $0 \times e \equiv 0$, $0 \odot e \equiv 0$, and $e - e \equiv 0$. For $0 \oplus e \equiv e$, $0 \times e \equiv 0$, and $0 \odot e \equiv 0$, we directly search for the constant 0. For $e \oplus e \equiv 0$ and $e - e \equiv 0$, the representation of computations in E shares the same common sub-expressions so that we do not need to compare whether two sub-expressions are the same or not when applying $\mathsf{Simply}_{\mathsf{Alg}}(E)$. Moreover, instead of considering only sub-expressions of the form $e \oplus e$ (resp. $e - e$), we search for two occurrences of the sub-expression e such that the operators on the path between the roots of two occurrences of e are all $\oplus$ (resp. $-$).

It is straightforward to verify the following proposition.

Proposition 4.9. *Given a program P and an observable set O, let $\overline{P}$ denote the program*

$$(x \leftarrow \overline{\mathcal{E}(x)};)_{x \in O}\ \mathtt{return};$$

where $\overline{\mathcal{E}(x)}$ is obtained from $\mathcal{E}(x)$ by applying $\mathsf{Simply}_{\mathsf{Dom}}(E)$, $\mathsf{Simply}_{\mathsf{Col}}(E)$, *and/or* $\mathsf{Simply}_{\mathsf{Alg}}(E)$; *then $[\![\overline{P}]\!]^{O}$ and $[\![P]\!]^{O}$ generate the same distribution over O.*

Example 4.10. Let us consider the above example, i.e., the observable set $\{x_1, x_2\}$, where

- $\mathcal{E}(x_1) = \Big(\mathsf{Sbox}\big((0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus r_2\Big) \oplus r_3$,
- $\mathcal{E}(x_2) = \Big(\mathsf{Sbox}\big((r_0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus r_2\Big) \oplus r_3$,

k is a private input variable, and r_0, r_1, r_2, r_3 are random variables. The type system in Figure 4 fails to prove $\vdash \{x_1, x_2\} : \tau_{\mathsf{uf}}$.

One can observe that $Z = \{r_2, r_3\}$ is collapsible with respect to $\{\mathcal{E}(x_1), \mathcal{E}(x_2)\}$, so by replacing $Z = \{r_2, r_3\}$ with a new random variable $\overline{Z}$, $\{\mathcal{E}(x_1), \mathcal{E}(x_2)\}$ can be simplified to

$$E_1 = \Big\{\mathsf{Sbox}\big((0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus \overline{Z}, \mathsf{Sbox}\big((r_0 \oplus ((k \oplus r_0) \oplus r_1)) \oplus r_1\big) \oplus \overline{Z}\Big\}.$$

By iteratively applying $\mathsf{Simply}_{\mathsf{Alg}}$ to E_1 using algebraic laws $r_0 \oplus r_0 \equiv 0$ and $r_1 \oplus r_1 \equiv 0$, we obtain

$$E_2 = \Big\{\mathsf{Sbox}\big(0 \oplus ((k \oplus r_0) \oplus 0)\big) \oplus \overline{Z}, \mathsf{Sbox}\big((k \oplus 0) \oplus 0\big) \oplus \overline{Z}\Big\}.$$

Since $0 \oplus e \equiv 0 \oplus e \equiv e$, by iteratively applying $\mathsf{Simply}_{\mathsf{Alg}}$ to E_2, we obtain

$$E_3 = \{\mathsf{Sbox}(k \oplus r_0) \oplus \overline{Z}, \mathsf{Sbox}(k) \oplus \overline{Z}\}.$$

Since r_0 is the dominant variable of $\mathsf{Sbox}(k \oplus r_0) \oplus \overline{Z}$ but does not occur in $\mathsf{Sbox}(k) \oplus \overline{Z}$, by applying $\mathsf{Simply}_{\mathsf{Dom}}$, we obtain $E_4 = \{r_0, \mathsf{Sbox}(k) \oplus \overline{Z}\}$. Now, $\overline{Z}$ becomes the dominant variable of $\mathsf{Sbox}(k) \oplus \overline{Z}$ but does not occur in r_0; by applying $\mathsf{Simply}_{\mathsf{Dom}}$ again, we obtain that $E_5 = \{r_0, \overline{Z}\}$, from which we can deduce $\vdash \{x_1, x_2\} : \tau_{\mathsf{uf}}$.

4.4 Type Inference Algorithm

In this sub-section, we present our type inference algorithm.

To prove that P is order-d secure, it is necessary to ensure that, for all size-d observable subsets $O \subseteq X_o$, P is O-**SI**. Evidently, exhaustive enumeration of $\binom{|X_o|}{d}$ subsets may not scale. To address this issue, the key idea is Proposition 2.1, which states that if the program P is O-**SI** (resp. O-uniform), then P is also O'-**SI** (resp. O'-uniform) for any subset $O' \subseteq O$. Therefore, the main

ALGORITHM 1: Type inference algorithm.

1 $\mathsf{PLS} := \emptyset$; λ :=empty_map; π :=empty_map;
2 **Function** $\mathsf{HOME}(P, X_p, X_k, X_r, X_o, d)$
3     $X_{\mathsf{check}} := \{x \in X_o \mid \mathsf{Var}(\mathcal{E}(x)) \nsubseteq X_p\}$;
4   **forall** $x \in X_{\mathsf{check}}$ **do**
5   **if** $\mathsf{Simply}_{\mathsf{Alg}}(\mathcal{E}(x)) \neq \mathcal{E}(x)$ **then**
6             $\lambda(x) := \mathsf{Simply}_{\mathsf{Alg}}(\mathcal{E}(x))$;
7             $\pi(x) := \mathsf{Dom}(\lambda(x))$;
8   **else** $\pi(x) := \mathsf{Dom}(\mathcal{E}(x))$;
9     $\mathsf{Explore}(\{(d, X_{\mathsf{check}})\})$;
10   **return** PLS;

11 **Function** $\mathsf{Explore}(\mathcal{Y})$
12   **forall** $(i, O) \in \mathcal{Y}$ **do**
13   Choose a subset $C_{i,O} \subseteq O$ in a topological order from leaf to root s.t. $|C_{i,O}| = i$;
14   **if** $\mathsf{Check}(\{C_{i,O}\}_{(i,O)\in\mathcal{Y}}) = \top$ **then**
15   **forall** $(i, O) \in \mathcal{Y},\ x \in O \setminus C_{i,O}$ *in a topological order from leaf to root* **do**
16   **if** $\mathsf{Check}(\{C_{i,O}\}_{(i,O)\in\mathcal{Y}}, \{x\}) = \top$ **then**
17                 $C_{i,O} := C_{i,O} \cup \{x\}$;
18   **else** $\mathsf{PLS} := \mathsf{PLS} \cup \{\bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}\}$;
19     $\mathcal{Y}' := \{(i, O) \in \mathcal{Y} \mid |O| > i \wedge i \neq 0\}$;
20   **if** $\mathcal{Y}' = \emptyset$ **then return**;
21   **forall** $(i, O) \in \mathcal{Y}', 0 \leq i_j \leq \min(i, |O \setminus C_{i,O}|)$ *s.t.* $\sum_{(i,O)\in\mathcal{Y}'} i_j \neq 0$ **do**
22         $\mathsf{Explore}((\mathcal{Y} \setminus \mathcal{Y}') \cup \bigcup_{(i,O)\in\mathcal{Y}'} \{(i - i_j, C_{i,O}), (i_j, O \setminus C_{i,O})\})$;
23   **return**;

24 **Function** $\mathsf{Check}(\{C_{i,O}\}_{(i,O)\in\mathcal{Y}}, Y = \emptyset)$
25   **if** $\vdash Y \cup \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O} : \tau$ *for some* $\tau \in \{\tau_{\mathtt{uf}}, \tau_{\mathtt{si}}\}$ *is valid* **then**
26   **return** $\top$;
27   **else if** $\vdash_{\mathsf{Simply}_{\mathsf{Dom}}} Y \cup \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O} : \tau$ *for some* $\tau \in \{\tau_{\mathtt{uf}}, \tau_{\mathtt{si}}\}$ *is valid* **then**
28   **return** $\top$;
29   **else if** $\vdash_{\mathsf{Simply}_{\mathsf{Col}}} Y \cup \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O} : \tau$ *for some* $\tau \in \{\tau_{\mathtt{uf}}, \tau_{\mathtt{si}}\}$ *is valid* **then**
30   **return** $\top$;
31   **return** $\bot$;

strategy is to find observable sets $\{O_i\}_{i=1}^n$ as large as possible such that P is O_i-**SI** for all $1 \leq i \leq n$, and for each size-d subset $O \subseteq X_o$, $O \subseteq O_i$ for some $1 \leq i \leq n$.

Our idea is formalized in Algorithm 1, where $\vdash O : \tau$ denotes the type inference without applying the transformations $\mathsf{Simply}_{\mathsf{Dom}}$ or $\mathsf{Simply}_{\mathsf{Col}}$; $\vdash_{\mathsf{Simply}_{\mathsf{Dom}}} O : \tau$ denotes the type inference aided with the transformation $\mathsf{Simply}_{\mathsf{Dom}}$; $\vdash_{\mathsf{Simply}_{\mathsf{Col}}} O : \tau$ denotes the type inference aided by both transformations $\mathsf{Simply}_{\mathsf{Dom}}$ and $\mathsf{Simply}_{\mathsf{Col}}$. Taking a program P; sets of public (X_p), private (X_k), random (X_r), and observable (X_o) variables; and the order d as inputs, the algorithm first initializes three data structures: PLS for storing all *potential* leaky observable sets, λ for storing the simplified computation of each variable, and π for storing the set of dominant variables of the (simplified) computation $\mathcal{E}(x)$ for each variable x.

At Line 3, Algorithm 1 computes the set X_{check} of observable variables whose computation involves either private or random variables. This allows to isolate the set of observable variables whose computation involves public input variables only. Hence, according to rule ($\textsc{Sid}_1$), it suffices to consider size-d subsets $O \subseteq X_o \setminus \{x \in X_o \mid \mathsf{Var}(\mathcal{E}(x)) \subseteq X_p\}$. At Lines 4 to 8, it simplifies the

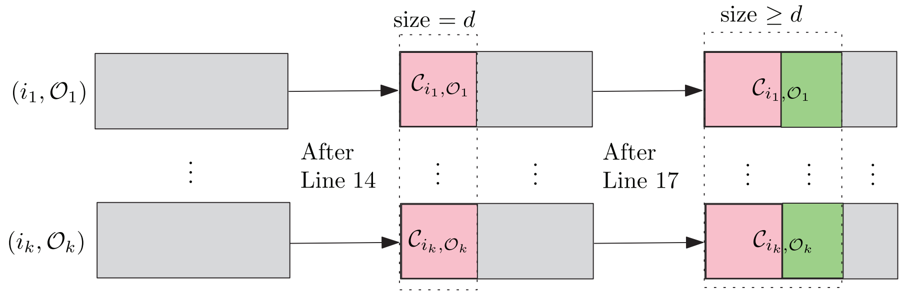

Fig. 5. Intuition of Algorithm 1.

computation $\mathcal{E}(x)$ for each variable $x \in X_{\mathsf{check}}$ by invoking $\mathsf{Simply}_{\mathsf{Alg}}$ and computes its dominant variables; the results are stored in λ and π for later use. After that, it invokes the function Explore with the set $\{(d, X_{\mathsf{check}})\}$ (Line 9). We assume that $|X_{\mathsf{check}}| \geq d$; otherwise, we can directly check whether $\vdash X_{\mathsf{check}} : \tau_{\mathsf{si}}$ is valid or not.

The function Explore is more involved. It aims at proving that for all pairs $(i, O) \in \mathcal{Y}$ and all possible subsets $O_i \subseteq O$ with size i, the type judgment $\vdash \bigcup_{(i,O)\in\mathcal{Y}} O_i : \tau_{\tau_{\mathsf{si}}}$ is valid. Taking a set $\mathcal{Y}$ of pairs (i, O) as input satisfies the following three properties:

(1) $\sum_{\{i|(i,O)\in\mathcal{Y}\}} i = d$, namely, the sum of orders' i for subsets O in $\mathcal{Y}$ is the target order d;
(2) $\biguplus\{O \mid (i, O) \in \mathcal{Y}\} = X_{\mathsf{check}}$, namely, the subsets O in $\mathcal{Y}$ form a partition of X_{check}; and
(3) $|O| \geq i$ for all $(i, O) \in \mathcal{Y}$, namely, there are at least i variables in O for each $(i, O) \in \mathcal{Y}$.

Remark that these properties are maintained and required to show the correctness and termination of our algorithm.

An illustration of the function Explore is given in Figure 5. The function Explore first chooses a size-i subset $C_{i,O} \subseteq O$ for each pair $(i, O) \in \mathcal{Y}$ in a topological order from leaf to root (Line 13). Then it checks whether the type judgment $\vdash \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O} : \tau$ for some $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}\}$ is valid or not by invoking the function Check (Line 14).

- If it is valid, i.e., the observable set $\bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}$ has distribution type τ_{uf} or τ_{si} (as shown in the middle part of Figure 5), then Explore iteratively tries to add the remaining observable variables x to $C_{i,O}$ for $x \in O \setminus C_{i,O}$ and $(i, O) \in \mathcal{Y}$ by invoking the function Check (Lines 15 to 17). The effect of this addition is shown in the right part of Figure 5.
- Otherwise, $\vdash \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O} : \tau$ for any $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}\}$ is invalid, and then $\bigcup_{(i,O)\in\mathcal{Y}} O_i$ is a potentially leaky set and is added to the set PLS (Line 18).

Finally, to cover $\bigcup_{(i,O)\in\mathcal{Y}} O_i$ for all possible size-i subsets $O_i \subseteq O$ and pairs $(i, O) \in \mathcal{Y}$, it remains to check the observable sets $\bigcup_{(i,O)\in\mathcal{Y}} O_i$, where there exists at least one pair $(i, O) \in \mathcal{Y}$ such that O_i contains at least one variable from $O \setminus C_{i,O}$. (Otherwise, $\bigcup_{(i,O)\in\mathcal{Y}} O_i \subseteq \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}$.) To do this, we first extract the pairs (i, O) such that $|O| > i$ and $i \neq 0$, i.e., $\mathcal{Y}' := \{(i, O) \in \mathcal{Y} \mid |O| > i \wedge i \neq 0\}$ at Line 19. If $\mathcal{Y}'$ is empty, then all the possible subsets $\bigcup_{(i,O)\in\mathcal{Y}} O_i$ are covered and Algorithm 1 terminates (Line 20). Otherwise, we partition all the pairs $(i, O) \in \mathcal{Y}'$ into pairs $(i - i_j, C_{i,O}), (i_j, O \setminus C_{i,O})$ for all combinations of values $0 \leq i_j \leq \min(i, |O \setminus C_{i,O}|)$ such that $\sum_{(i,O)\in\mathcal{Y}'} i_j \neq 0$. The condition $\sum_{(i,O)\in\mathcal{Y}'} i_j \neq 0$ is used to avoid the case $\bigcup_{(i,O)\in\mathcal{Y}} O_i \subseteq \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}$. For each such combination of values, the partitioned pairs $\{(i - i_j, C_{i,O}), (i_j, O \setminus C_{i,O}) \mid (i, O) \in \mathcal{Y}'\}$ together with the pairs $\{(i, O) \in \mathcal{Y} \mid |O| = i \vee i = 0\}$ (i.e., $\mathcal{Y} \setminus \mathcal{Y}'$) are checked by recursively calling the function Explore. It is easy to observe that the recursion maintains the above three properties.

The function Check first verifies whether $\vdash Y \cup \bigcup_{(i,O)\in\mathcal{Y}} O_i : \tau$ for some $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}\}$ is valid, which may be aided with data structures λ and π (Line 25). If it is valid, $\top$ is returned (Line 26). Otherwise, it is verified with the additional transformation $\text{Simply}_{\text{Dom}}$ (Line 27). If it still fails, $\vdash Y \cup \bigcup_{(i,O)\in\mathcal{Y}} O_i{:}\tau$ for some $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}\}$ is checked using the additional transformation $\text{Simply}_{\text{Col}}$ on the expressions yielded by $\text{Simply}_{\text{Dom}}$ (Line 29). Once $\vdash Y \cup \bigcup_{(i,O)\in\mathcal{Y}} O_i : \tau$ for some $\tau \in \{\tau_{\mathsf{uf}}, \tau_{\mathsf{si}}\}$ is derived, Check returns $\top$ (Lines 28 and 30). If all of these steps fail, $\perp$ is returned (Line 31). Notice that during the above type inference, once $\vdash Y \cup \bigcup_{(i,O)\in\mathcal{Y}} O_i : \tau_{\mathsf{lk}}$ becomes valid, Check also returns $\perp$. Moreover, in order to avoid recomputing $\text{Simply}_{\text{Dom}}$ and $\text{Simply}_{\text{Col}}$, the sequence of applied transformations is recorded, and the simplified expressions are cached. When the function Check is invoked at Line 16, i.e., Y is nonempty, we first check whether the recorded sequence of applied transformations is still legal. If it is still applicable, we will reuse the simplified expressions and apply $\text{Simply}_{\text{Dom}}$ and/or $\text{Simply}_{\text{Col}}$ to $\mathcal{E}(x)$ as well. Otherwise, the function Check immediately returns $\perp$.

Remark that the type inference rules are applied in the order of increasing complexities of checking the premises while preferring τ_{uf} over τ_{si}. We also remark that the choice of the subsets at Line 13 and the variable x at Line 15 may have significant impact on the performance. We choose variables from leaf to root following the order of the size of the defining computation, in light of Rule (Sɪᴅ$_2$) in Figure 4.

The procedure terminates as we only partition pairs $(i,O) \in \mathcal{Y}$ such that $|O| > i$ and $i \neq 0$ and the sizes of $C_{i,O}$ and $O \setminus C_{i,O}$ in partitioned pairs $(i - i_j, C_{i,O})$ and $(i_j, O \setminus C_{i,O})$ eventually become smaller and smaller in recursive calls until $|O| = i$ or $i = 0$. (Note that we keep pairs of the form $(0, O)$ in the worklist for simplifying presentation. They are indeed removed in our implementation.)

Theorem 4.11. *P is order-d secure if $\mathsf{PLS} = \emptyset$. Moreover, if P is O-leaky for $O \subseteq X_{\mathsf{check}}$ with $|O| = d$, then $O \in \mathsf{PLS}$.*

Note that the reverse of Theorem 4.11 may not hold. To prove Theorem 4.11, we start with the following lemmas. First, we show that the above three properties always hold.

Lemma 4.12. *In Algorithm 1, at each call Explore($\mathcal{Y}$), the following three properties hold:*

(1) $\sum_{\{(i,O)\in\mathcal{Y}\}} i = d$;
(2) $\biguplus\{O \mid (i,O) \in \mathcal{Y}\} = X_{\mathsf{check}}$; *and*
(3) $|O| \geq i$ *for all* $(i,O) \in \mathcal{Y}$.

Proof. Let $\mathcal{Y}_\ell$ denote the parameter $\mathcal{Y}$ at the ℓ^{th} call of Explore. Let us apply induction on ℓ. The base case $\ell = 1$ immediately follows from the fact that $\mathcal{Y}_1 = \{(d, X_{\mathsf{check}})\}$ (note that we assumed $|X_{\mathsf{check}}| \geq d$). It remains to prove the inductive step. Suppose the result holds at $\ell > 1$ and $\mathcal{Y}_{\ell+1} = (\mathcal{Y}_\ell \setminus \mathcal{Y}') \cup \{(i - i_j, C_{i,O}), (i_j, O \setminus C_{i,O}) \mid (i,O) \in \mathcal{Y}'\}$, where $\mathcal{Y}' = \{(i,O) \in \mathcal{Y}_\ell \mid |O| > i \wedge i \neq 0\}$.

- By applying the induction hypothesis, we get that $\sum\{i \mid (i,O) \in \mathcal{Y}_\ell\} = d$. Since

$$\begin{aligned}\textstyle\sum\{i \mid (i,O) \in \mathcal{Y}_{\ell+1}\} &= \textstyle\sum\{i \mid (i,O) \in \mathcal{Y}_\ell \setminus \mathcal{Y}'\} + \sum\{i - i_j, i_j \mid (i,O) \in \mathcal{Y}'\} \\ &= \textstyle\sum\{i \mid (i,O) \in \mathcal{Y}_\ell \setminus \mathcal{Y}'\} + \sum\{i \mid (i,O) \in \mathcal{Y}'\} \\ &= \textstyle\sum\{i \mid (i,O) \in \mathcal{Y}_\ell\},\end{aligned}$$

  we conclude the proof of Item (1).

- By applying the induction hypothesis, we get that $\biguplus\{O \mid (i,O) \in \mathcal{Y}_\ell\} = X_{\mathsf{check}}$. Since

$$\begin{aligned}\biguplus\{O \mid (i,O) \in \mathcal{Y}_{\ell+1}\} &= \biguplus\{O \mid (i,O) \in \mathcal{Y}_\ell \setminus \mathcal{Y}'\} \uplus \biguplus\{C_{i,O}, O \setminus C_{i,O} \mid (i,O) \in \mathcal{Y}'\}\\ &= \biguplus\{O \mid (i,O) \in \mathcal{Y}_\ell \setminus \mathcal{Y}'\} \uplus \biguplus\{O \mid (i,O) \in \mathcal{Y}'\}\\ &= \biguplus\{O \mid (i,O) \in \mathcal{Y}_\ell\},\end{aligned}$$

  we conclude the proof of Item (2).
- By applying the induction hypothesis, $|O| \geq i$ for all $(i,O) \in \mathcal{Y}_\ell \setminus \mathcal{Y}'$. For each pair $(i,O) \in \mathcal{Y}_\ell$, according to Lines 13 and 17, $|C_{i,O}| \geq i$, and hence, $|C_{i,O}| \geq i - i_j$. Since $0 \leq i_j \leq \min(i, |O \setminus C_{i,O}|)$, we get that $|O \setminus C_{i,O}| \geq i_j$. We conclude the proof of Item (3). □

We now prove the termination of Algorithm 1.

Lemma 4.13. *Algorithm 1 always terminates.*

Proof. It suffices to show that the recursive procedure call of Explore always terminates. Let $\mathcal{Y}_\ell$ denote the parameter $\mathcal{Y}$ at the ℓ^{th} call of Explore. By Lemma 4.12(3), $|O| \geq i$ for all $(i,O) \in \mathcal{Y}_\ell$.

- If $|O| = i$ or $i = 0$ for all $(i,O) \in \mathcal{Y}_\ell$, then $\mathcal{Y}' = \emptyset$. In this case, Explore will not be called at Line 22 during the ℓ^{th} call. Hence, Algorithm 1 terminates.
- Otherwise, there are some $(i,O) \in \mathcal{Y}_\ell$ such that $|O| > i$ and $i \neq 0$. Then, such pairs (i,O) are always partitioned into $(i - i_j, C_{i,O})$ and $(i_j, O \setminus C_{i,O})$. Since $\sum_{(i,O)\in\mathcal{Y}'} i_j \neq 0$, by Lemma 4.12(1 and 2), there exists $\ell' > \ell$ for $\mathcal{Y}_{\ell'}$ such that $|O| = i$ or $i = 0$ for all $(i,O) \in \mathcal{Y}_{\ell'}$. Hence, Algorithm 1 always terminates. □

We show the soundness of Algorithm 1.

Lemma 4.14. *For every subset $O \subseteq X_{\mathsf{check}}$ such that $|O| = d$, O is covered by Algorithm 1; namely, either O is added into* PLS *or there exists a subset O' such that $\vdash O : \tau_{\mathtt{si}}$ is valid and $O \subseteq O'$.*

Proof. Given a set $\mathcal{Y}$ of pairs, let $Cover(\mathcal{Y})$ denote the set of subsets $O \subseteq X_{\mathsf{check}}$ such that $|O| = d$ and O contains i elements of O' for each pair $(i,O') \in \mathcal{Y}$. It suffices to show that for every call Explore($\mathcal{Y}$), each subset $O \in Cover(\mathcal{Y})$ is covered.

Let $\mathcal{Y}_\ell$ denote the parameter $\mathcal{Y}$ at the ℓ^{th} call of Explore. We apply induction on ℓ, where the base case is the largest ℓ. Note that such ℓ exists by Lemma 4.13.

Base case. The base case is the largest ℓ such that $\mathcal{Y}' = \emptyset$ at the ℓ^{th} call of Explore. Since $\mathcal{Y}' = \{(i,O') \in \mathcal{Y}_\ell \mid |O'| > i \wedge i \neq 0\} = \emptyset$, by Lemma 4.12(3), $|O'| = i$ or $i = 0$ for all $(i,O') \in \mathcal{Y}_\ell$. By Lemma 4.12(1 and 2), $Cover(\mathcal{Y})$ is a singleton set. Suppose $Cover(\mathcal{Y}) = \{O\}$; then O is covered. Indeed, either $\vdash O : \tau_{\mathtt{si}}$ is valid or O is added into PLS.

Inductive step. There exists some pair $(i,O') \in \mathcal{Y}_\ell$ such that $|O'| > i$ and $i \neq 0$. For every subset $O \in Cover(\mathcal{Y}_\ell)$, either $O \subseteq \bigcup_{(i,O')\in\mathcal{Y}_\ell} C_{i,O'}$ or $O \nsubseteq \bigcup_{(i,O')\in\mathcal{Y}_\ell} C_{i,O'}$.

- If $O \subseteq \bigcup_{(i,O')\in\mathcal{Y}_\ell} C_{i,O'}$, then O is covered. Indeed, $\vdash \bigcup_{(i,O')\in\mathcal{Y}_\ell} C_{i,O'} : \tau_{\mathtt{si}}$ is valid.
- Otherwise, $O \nsubseteq \bigcup_{(i,O')\in\mathcal{Y}_\ell} C_{i,O'}$, and then O contains at least one variable from $O' \setminus C_{i,O'}$ for some pair $(i,O') \in \mathcal{Y}_\ell$, i.e., $O \cap (O' \setminus C_{i,O'}) \neq \emptyset$. There must exist a combination of values $i_j : 0 \leq i_j \leq \min(i, |O' \setminus C_{i,O'}|)$ for $(i,O') \in \mathcal{Y}'$ such that

$$O \in cover((\mathcal{Y}_\ell \setminus \mathcal{Y}') \cup \{(i - i_j, C_{i,O'}), (i_j, O' \setminus C_{i,O'}) \mid (i,O') \in \mathcal{Y}'\}).$$

  By applying the induction hypothesis, the subset O is covered.

We complete the proof. □

Proof of Theorem 4.11. If $\mathsf{PLS} = \emptyset$, then by Lemma 4.14, $\vdash O : \tau_{\mathtt{si}}$ is valid for every size-d subset $O \subseteq X_{\mathsf{check}}$. Hence, P is order-d secure.

On the other hand, for every size-d subset $O \subseteq X_{\text{check}}$, if P is O-leaky, then $\vdash O : \tau_{\mathtt{si}}$ is not valid. By Lemma 4.14, all the size-d subsets $O \subseteq X_{\text{check}}$ are covered by Algorithm 1, and hence, O is added into PLS. □

Example 4.15. We demonstrate Algorithm 1 on the motivating example (cf. Section 3.1) for $d = 2$. First of all, $X_{\text{check}} = X_o = \{x', A, y_0, \ldots, y_5, r, r'\}$ as $X_p = \emptyset$. After applying the transformation $\text{Simply}_{\text{Alg}}$, λ and π are given below:

$$\lambda(y_4) = r' \oplus k, \qquad \lambda(y_5) = (r' \oplus k) - (r' \oplus r),$$
$$\lambda(A) = \Big(r' \oplus k) - (r' \oplus r)\Big) \oplus \Big((((k \oplus r) \oplus r') - r') \oplus (k \oplus r)\Big).$$

$$\begin{array}{lllll} \pi(x') = \{r\}, & \pi(r) = \{r\}, & \pi(r') = \{r'\}, & \pi(y_0) = \{r, r'\}, & \pi(y_1) = \{r\}, \\ \pi(y_2) = \emptyset, & \pi(y_3) = \{r, r'\}, & \pi(y_4) = \{r'\}, & \pi(y_5) = \{r\}, & \pi(A) = \emptyset. \end{array}$$

HOME invokes Explore($\{(2, X_o)\}$). Suppose Explore chooses $\{r, r'\}$ at Line 13, i.e., $C_{2,X_o} = \{r, r'\}$; then $\vdash C_{2,X_o} : \tau_{\mathtt{uf}}$ is valid; namely, Check($\{C_{2,X_o}\}$) will return $\top$. The loop at Lines 15 to 17 will iteratively test $x', y_0, \ldots, y_5, A$. Among them, only y_3 can be added into C_{2,X_o} according to rule (Sɪᴅ$_2$). Now we can deduce that all size-2 subsets $O \subseteq C_{2,X_o} = \{r, r', y_3\}$ have type $\tau_{\mathtt{uf}}$ or $\tau_{\mathtt{si}}$.

It is easy to see that $\mathcal{Y}' = \{(2, X_o)\}$, as $|X_o| > 2$. Therefore, at Line 22, the following two procedure calls will be made:

- **Call$_1$:** Explore($\mathcal{Y}_1$), where $\mathcal{Y}_1 = \{(0, \{r, r', y_3\}), (2, \{x', A, y_0, y_1, y_2, y_4, y_5\})\}$.
- **Call$_2$:** Explore($\mathcal{Y}_2$), where $\mathcal{Y}_2 = \{(1, \{r, r', y_3\}), (1, \{x', A, y_0, y_1, y_2, y_4, y_5\})\}$.

For **Call$_1$**, suppose Explore chooses $\{x', y_0\}$ at Line 13, i.e., $C_{2,\{x',A,y_0,y_1,y_2,y_4,y_5\}} = \{x', y_0\}$. We derive $\vdash \{x', y_0\} : \tau_{\mathtt{uf}}$ by applying rule (Rᴜᴅ) with $O_1 = \{x'\}$ and $O_2 = \{y_0\}$. The loop at Lines 15 to 17 will add y_1 into $C_{2,\{x',A,y_0,y_1,y_2,y_4,y_5\}}$, as replacing the common sub-expression $k \oplus r$ in $\{\mathcal{E}(x'), \mathcal{E}(y_0), \mathcal{E}(y_1)\}$ with r using the transformation $\text{Simply}_{\text{Dom}}$, i.e., $\mathcal{E}(x')$, $\mathcal{E}(y_0)$, and $\mathcal{E}(y_1)$, respectively, becoming r, $r \oplus r'$, $(r \oplus r') - r'$, allows us to derive $\vdash \{x', y_0, y_1\} : \tau_{\mathtt{si}}$. By replaying this transformation on $\mathcal{E}(y_2)$, y_2 can also be added into $C_{2,\{x',A,y_0,y_1,y_2,y_4,y_5\}}$, which becomes $\{x', y_0, y_1, y_2\}$. However, y_4, y_5, and A cannot be added into $C_{2,\{x',A,y_0,y_1,y_2,y_4,y_5\}}$.

For **Call$_2$**, suppose Explore chooses $\{r\}$ from $\{r, r', y_3\}$ and x' from $\{x', A, y_0, y_1, y_2, y_4, y_5\}$ at Line 13. We cannot derive any type judgment $\vdash \{r, x'\} : \tau$ for $\tau \in \mathcal{T}$. So $\{r, x'\}$ is added to PLS. Finally,

$$\mathsf{PLS} = \left\{ \begin{array}{l} \{r, r', x', y_2, y_3, y_4, y_5\} \times \{A\} \cup \ \{r', y_2\} \times \{y_4\} \ \cup \\ \quad \{r, x', y_0, y_1, y_2\} \times \{y_5\} \quad \cup \{r\} \times \{x', y_1, y_2\} \cup \{y_2, y_3\} \end{array} \right\}.$$

5 MODEL-COUNTING- AND PATTERN-MATCHING-BASED METHODS

We propose in this section two model-counting-based methods (cf. Sections 5.1 and 5.2) for resolving potential leaky observable sets prescribed by type inference algorithm. Generally, model counting is very costly, so we propose a complementary pattern-matching-based method (cf. Section 5.3) to efficiently resolve potential leaky observable sets from known sets, avoiding a vast amount of model-counting usage.

5.1 SMT-Based Method

We first lift the SMT-based method [60] from first order to higher order.

Recall that P is O-leaky iff $[\![P]\!]^O_{\eta_1} \neq [\![P]\!]^O_{\eta_2}$ for some pair $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$. Let $O = \{x_1, \ldots, x_m\}$. For every valuation $\eta \in \Theta$ and tuple of values $(c_1, \ldots, c_m) \in \mathbb{I}^m$, let $\sharp_\eta(x_1 = c_1, \ldots, x_m = c_m)$ denote the number of assignments $\eta_r : X_r \to \mathbb{I}$ such that for all $1 \leq j \leq m$, $\mathcal{E}(x_j)$ evaluates to c_j under

η and η_r. Then, $\mathcal{O}$-leaky can be characterized as the following logical formula:

$$\Omega^{\mathcal{O}} := \exists(\eta_1,\eta_2) \in \Theta^2_{=X_p} \exists(c_1,\ldots,c_m) \in \mathbb{I}^m. \\ \left(\sharp_{\eta_1}(x_1 = c_1,\ldots,x_m = c_m) \neq \sharp_{\eta_2}(x_1 = c_1,\ldots,x_m = c_m)\right). \tag{1}$$

Proposition 5.1. *$\Omega^{\mathcal{O}}$ is satisfiable iff P is $\mathcal{O}$-leaky.*

Proof. The program P is $\mathcal{O}$-leaky iff the following formula holds:

$$\exists(\eta_1,\eta_2) \in \Theta^2_{=X_p} \exists(c_1,\ldots,c_m) \in \mathbb{I}^m.[\![P]\!]^{\mathcal{O}}_{\eta_1}(c_1,\ldots,c_m) \neq [\![P]\!]^{\mathcal{O}}_{\eta_2}(c_1,\ldots,c_m).$$

Since $[\![P]\!]^{\mathcal{O}}_{\eta}(c_1,\ldots,c_m) = \frac{\sharp_\eta(x_1=c_1,\ldots,x_m=c_m)}{2^{\kappa\times|X_r|}}$ for $\eta \in \{\eta_1,\eta_2\}$, then the program P is $\mathcal{O}$-leaky iff the following formula holds:

$$\exists(\eta_1,\eta_2) \in \Theta^2_{=X_p} \exists(c_1,\ldots,c_m) \in \mathbb{I}^m.\frac{\sharp_{\eta_1}(x_1 = c_1,\ldots,x_m = c_m)}{2^{\kappa\times|X_r|}} \neq \frac{\sharp_{\eta_2}(x_1 = c_1,\ldots,x_m = c_m)}{2^{\kappa\times|X_r|}}.$$

The result follows immediately. □

We further encode $\Omega^{\mathcal{O}}$ as a first-order logic formula that can be solved by SMT solvers (e.g., Z3 [50]). Suppose $\mathcal{E}(x_j) = e_j$ for $1 \leq j \leq m$; let $E_{\mathcal{O}} = E^1 \uplus E^2$ with $E^1 = \{e \mid \mathsf{Var}(e) \cap X_k \neq \emptyset\}$, and $E^2 = \{e \mid \mathsf{Var}(e) \cap X_k = \emptyset\}$. We define the first-order logic formula $\Psi^{\mathcal{O}}$ as

$$\Psi^{\mathcal{O}} := \begin{pmatrix} \left(\bigwedge_{e\in E^1} \bigwedge_{f:\mathsf{RVar}(e)\to\mathbb{I}}(\Theta_{e,f} \wedge \Theta'_{e,f})\right) \\ \bigwedge \\ \left(\bigwedge_{e\in E^2} \bigwedge_{f:\mathsf{RVar}(e)\to\mathbb{I}} \Theta_{e,f}\right) \\ \bigwedge \\ \left(\Theta_{v2i} \wedge \Theta'_{v2i} \wedge \Theta_{\neq}\right) \end{pmatrix}, \text{where}$$

- **Program logic ($\Theta_{e,f}$ and $\Theta'_{e,f}$):** for every expression $e = \mathcal{E}(x) \in E_{\mathcal{O}}$ denoting the computation of the variable x, and for every function $f : \mathsf{RVar}(e) \to \mathbb{I}$ that enumerates an assignment of the random variables, the logical formula $\Theta_{e,f}$ encodes the expression e into a first-order logic formula and asserts that the value of e is equal to a fresh variable x_f with all the random variables r instantiated by the concrete values $f(r)$. For instance, consider $e = (k \wedge r_1) \vee r_2$ and the function f with $f(r_1) = 1$ and $f(r_2) = 0$; then $\Theta_{e,f}$ is the logic formula $x_f = (k \wedge 1) \vee 0$. (Note there are $2^{|\mathsf{RVar}(e)|}$ distinct conjuncts, each of which corresponds to one possible assignment of the random variables, but all of which share the variables from $X_p \cup X_k$.)
  $\Theta'_{e,f}$ is similar to $\Theta_{e,f}$ except that the variables x_f and $k \in X_k$ in $\Theta_{e,f}$ are replaced by fresh variables x'_f and k', respectively. For instance, consider $e = (k \wedge r_1) \vee r_2$ with $k \in X_k$ and the function f with $f(r_1) = 1$ and $f(r_2) = 0$; then $\Theta'_{e,f}$ is the logic formula $x'_f = (k' \wedge 1) \vee 0$. Note that for every $e \in E^2$, we do not construct $\Theta'_{e,f}$, as $e \in E^2$ does not have any private variable $k \in X_k$ and hence $\Theta'_{e,f}$ would be the same as $\Theta_{e,f}$. For example, consider $e = (p \wedge r_1) \vee r_2$ with $p \in X_p$ and the function f with $f(r_1) = 1$ and $f(r_2) = 0$; then both $\Theta_{e,f}$ and $\Theta'_{e,f}$ are $x_f = (p \wedge 1) \vee 0$.
- **Vector to integer (Θ_{v2i} and Θ'_{v2i}):** Θ_{v2i} asserts that for every function $f : \bigcup_{e\in E_{\mathcal{O}}} \mathsf{RVar}(e) \to \mathbb{I}$ that enumerates an assignment of the random variables, a fresh integer variable I_f is 1 if $x_{i,f} = c_i$ holds for every variable $x_i \in \mathcal{O} = \{x_1,\ldots,x_m\}$, otherwise 0, where the fresh variables c_is in Θ_{v2i} are identical for all the functions fs. By doing so, we enumerate all the possible assignments of random variables and then count the number of assignments fs of random variables under which $(e_1,\ldots,e_m)$ evaluate $(c_1,\ldots,c_m)$ when

$$\begin{pmatrix} \big(y_{000} = (k \oplus 0) \oplus 0\big) \wedge \big(y'_{000} = (k' \oplus 0) \oplus 0\big) \wedge \big(y_{001} = (k \oplus 1) \oplus 0\big) \wedge \big(y'_{001} = (k' \oplus 1) \oplus 0\big) \wedge \\ \big(y_{010} = (k \oplus 0) \oplus 1\big) \wedge \big(y'_{010} = (k' \oplus 0) \oplus 1\big) \wedge \big(y_{011} = (k \oplus 1) \oplus 1\big) \wedge \big(y'_{011} = (k' \oplus 1) \oplus 1\big) \wedge \end{pmatrix}$$

$$\Big((y_{300} = 0 \oplus 0) \wedge (y_{301} = 0 \oplus 1) \wedge (y_{310} = 1 \oplus 0) \wedge (y_{311} = 1 \oplus 1) \wedge \Big)$$

$$\begin{pmatrix} \big(I_{00} = (y_{000} = c_1 \wedge y_{300} = c_2)\ ?\ 1 : 0\big) \wedge \big(I_{01} = (y_{001} = c_1 \wedge y_{301} = c_2)\ ?\ 1 : 0\big) \wedge \\ \big(I_{10} = (y_{010} = c_1 \wedge y_{310} = c_2)\ ?\ 1 : 0\big) \wedge \big(I_{11} = (y_{011} = c_1 \wedge y_{311} = c_2)\ ?\ 1 : 0\big) \wedge \end{pmatrix}$$

$$\begin{pmatrix} \big(I'_{00} = (y'_{000} = c_1 \wedge y_{300} = c_2)\ ?\ 1 : 0\big) \wedge \big(I'_{01} = (y'_{001} = c_1 \wedge y_{301} = c_2)\ ?\ 1 : 0\big) \wedge \\ \big(I'_{10} = (y'_{010} = c_1 \wedge y_{310} = c_2)\ ?\ 1 : 0\big) \wedge \big(I'_{11} = (y'_{011} = c_1 \wedge y_{311} = c_2)\ ?\ 1 : 0\big) \wedge \end{pmatrix}$$

$$\Big((I_{00} + I_{01} + I_{10} + I_{11}) \neq (I'_{00} + I'_{01} + I'_{10} + I'_{11})\Big)$$

Fig. 6. The SMT encoding $\Psi^{\{y_0, y_3\}}$.

variables $x \in X_p \cup X_k$ take some concrete values and random variables take concrete values from f. Intuitively, if there are two distinct functions f_1 and f_2 such that the values of x_fs are identical (i.e., $x_{i,f} = c_i$ holds for every variable $x_i \in O$ and $f \in \{f_1, f_2\}$) when the input variables $X_p \cup X_k$ are 0, we deduce that there are two assignments of random variables under which $(e_1, \ldots, e_m)$ evaluate $(c_1, \ldots, c_m)$ when the input variables $X_p \cup X_k$ are 0. Formally,

$$\Theta_{v2i} := \bigwedge_{f:\bigcup_{e \in E_O} \mathsf{RVar}(e) \to \mathbb{I}} \Big(I_f = \big((x_{1f} = c_1 \wedge \cdots \wedge x_{mf} = c_m)\ ?\ 1 : 0\big)\Big).$$

Θ'_{v2i} is similar to Θ_{v2i} except that I_f is replaced by I'_f, and x_f is replaced by x'_f for all $x \in O$ such that $\mathcal{E}(x) \in E^1$. Note that $k' \in X_k$ may have a different value than k, but $x \in X_p$ has the same value in Θ_{v2i} and Θ'_{v2i}. This conforms to $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$ in Equation (1).

- **Different sums ($\Theta_{\neq}$)**: It asserts that two sums of assignments fs of random variables for variables $(k)_{k \in X_k}$ and $(k')_{k \in X_K}$ (i.e., integers I_f and I'_f) differ. This conforms to $\big(\sharp_{\eta_1}(x_1 = c_1, \ldots, x_m = c_m) \neq \sharp_{\eta_2}(x_1 = c_1, \ldots, x_m = c_m)\big)$ in Equation (1). Formally,

$$\Theta_{\neq} := \sum_{f:\bigcup_{e \in E_O} \mathsf{RVar}(e) \to \mathbb{I}} I_f \neq \sum_{f:\bigcup_{e \in E_O} \mathsf{RVar}(e) \to \mathbb{I}} I'_f,$$

where $\sum_{f:\bigcup_{e \in E_O} \mathsf{RVar}(e) \to \mathbb{I}} I_f$ is the sum of all the assignments fs of random variables for $(k)_{k \in X_k}$, and $\sum_{f:\bigcup_{e \in E_O} \mathsf{RVar}(e) \to \mathbb{I}} I'_f$ is the sum of all the assignments fs of random variables for $(k')_{k \in X_k}$.

Overall, the logical formula Ω^O is satisfiable iff there exist a pair of valuations $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$ and an assignment of variables $(c_1, \ldots, c_m)$ such that the sums of assignments fs of random variables for variables $(k)_{k \in X_k}$ and $(k')_{k \in X_K}$ under which the expressions $(e_1, \ldots, e_m)$ evaluate to $(c_1, \ldots, c_m)$ are different.

It is straightforward to get the following proposition.

Proposition 5.2. *Ω^O is satisfiable iff Ψ^O is satisfiable, where the size of Ψ^O is exponential in the number of (bits of) random variables.*

By Proposition 5.1 and Proposition 5.2, we get that:

Corollary 5.3. *Ψ^O is satisfiable iff P is O-leaky.*

Example 5.4. Let us consider the observable set $\{y_0, y_3\}$ in the motivating example (cf. Section 3.1). Recall that $\mathcal{E}(y_0) = (k \oplus r) \oplus r'$ and $\mathcal{E}(y_3) = r' \oplus r$. In this case, $E^1 = \{\mathcal{E}(y_0)\}$ and $E^2 = \{\mathcal{E}(y_3)\}$. For clarity, we only show the case when all variables are Boolean. The SMT formula $\Psi^{\{y_0, y_3\}}$ is shown in Figure 6 .

ALGORITHM 2: A brute-force algorithm

Function BFEnum($P, O = \{x_1, \ldots, x_m\}$)
  forall $\eta_p : \bigcup_{x \in O} X_p \cap \mathsf{Var}(\mathcal{E}(x)) \rightarrow \mathbb{I}$ **do**
    $D_1 := \lambda(c_1, \ldots, c_m) \in \mathbb{I}^m.0$;
    $b :=$ `false`;
    forall $\eta_k : \bigcup_{x \in O} X_k \cap \mathsf{Var}(\mathcal{E}(x)) \rightarrow \mathbb{I}$ **do**
      $D_2 := \lambda(c_1, \ldots, c_m) \in \mathbb{I}^m.0$;
      if $b =$ `false` **then**
        $D_1 :=$Counting(P, O, η_p, η_k);
        $b :=$ `true`;
      else
        $D_2 :=$Counting(P, O, η_p, η_k);
        if $D_1 \neq D_2$ **then return** SAT;
  return UNSAT;

Function Counting($P, O = \{x_1, \ldots, x_m\}, \eta_p, \eta_k$)
  forall $\eta_r : \bigcup_{x \in O} \mathsf{RVar}(\mathcal{E}(x)) \rightarrow \mathbb{I}$ **do**
    $D[\mathcal{E}_{\eta_p, \eta_k, \eta_r}(x_1), \ldots, \mathcal{E}_{\eta_p, \eta_k, \eta_r}(x_m)] + +$;
  return D;

- The first two lines correspond to the logical formulas $\Theta_{\mathcal{E}(y_0),f}$ and $\Theta'_{\mathcal{E}(y_0),f}$, which enumerates all the possible functions $f : \{r, r'\} \rightarrow \{0, 1\}$ of the random variables. For instance, the conjunct $y_{000} = (k \oplus 0) \oplus 0$ (resp. $y_{001} = (k \oplus 1) \oplus 0$) asserts that the computation $\mathcal{E}(y_0)$ is equal to the fresh variable y_{000} (resp. y_{001}) when the random variables r and r' are assigned by 0 (resp. 0 and 1). y'_{000} and y'_{001} are the same as y_{000} and y_{001} except that the private input variable k is replaced by k'.
- The third line corresponds to the logical formulas $\Theta_{\mathcal{E}(y_3),f}$, which enumerates all the possible functions $f : \{r, r'\} \rightarrow \{0, 1\}$ of the random variables. Since the computation $\mathcal{E}(y_3) = r' \oplus r$ does not involve any private input variable, we omit the logical formulas $\Theta'_{\mathcal{E}(y_3),f}$ that are the same as $\Theta_{\mathcal{E}(y_3),f}$.
- The next four lines correspond to the logical formulas Θ_{v2i} and Θ'_{v2i}. For instance, the conjunct $I_{00} = (y_{000} = c_1 \wedge y_{300} = c_2)\ ?\ 1 : 0$ asserts that the fresh integer variable I_{00} (corresponding to the function f with $f(r) = f(r') = 0$) is 1 if y_{000} (i.e., the computation $\mathcal{E}(y_0)$ under the function f with $f(r) = f(r') = 0$) is c_1 and y_{300} (i.e., the computation $\mathcal{E}(y_3)$ under the function f with $f(r) = f(r') = 0$) is c_2.
- The last one corresponds to the logical formula $\Theta_{\neq}$. For instance, the conjunct $(I_{00} + I_{01} + I_{10} + I_{11})$ sums up the numbers of functions $f : \{r, r'\} \rightarrow \{0, 1\}$ of the random variables such that the computations $\mathcal{E}(y_1)$ and $\mathcal{E}(y_3)$ evaluate to c_1 and c_2, respectively.

$\Psi^{\{y_0, y_3\}}$ is satisfiable. For instance, when $k = 1$, $k' = 0$, and $c_1 = c_2 = 0$, we can see that $I_{00} + I_{01} + I_{10} + I_{11} = 0$ and $I_{00} + I_{01} + I_{10} + I_{11} = 1 + 0 + 0 + 1 = 2$, which is a witness of $\Psi^{\{y_0, y_3\}}$. This implies that the program is $\{y_0, y_3\}$-leaky.

5.2 Brute-Force Method

The brute-force method (cf. Algorithm 2) enumerates all possible valuations and then computes corresponding distributions again by enumerating the assignments of random variables.

Proposition 5.5. *Ω^O is satisfiable iff Algorithm 2 returns* SAT.

The complexity of Algorithm 2 is exponential in the number of (bits of) variables in computations $(\mathcal{E}(x))_{x\in O}$, so it would experience significant performance degradation when facing a large number of variables. We propose a GPU-accelerated parallel algorithm to boost the performance (cf. Section 6.1).

5.3 Method Based on Pattern Matching

In order to avoid (costly) model counting, we propose a novel pattern-matching-based method, which allows to resolve potential leaky observable sets more efficiently. This idea comes from the observation that cryptographic programs usually have very similar blocks and many observable sets share common observable variables. As a warmup, let us first consider two observable sets $\{x, y\}$ and $\{x', y'\}$, where $\mathcal{E}(x) = r, \mathcal{E}(y) = k \oplus r, \mathcal{E}(x') = r', \mathcal{E}(y) = k \oplus r'$; k is a private input; and r, r' are two random variables. Then $\{\mathcal{E}(x), \mathcal{E}(y)\}$ and $\{\mathcal{E}(x'), \mathcal{E}(y')\}$ are equivalent up to renaming of random variables; thus, observable sets $\{x, y\}$ and $\{x', y'\}$ have same distribution type.

Based on this observation, we propose a pattern-matching method for inferring distribution types of observable sets O from observable sets O' whose distribution types are known. Before formalizing this idea, we first introduce type-respecting bijection functions.

Given a bijective function $f : X \to X$, the function f is *type respecting* if for every $x \in X$, $f(x)$ is public (resp. private and random) iff x is public (resp. private and random).

Definition 5.6. Two sets of computations E and E' are *isomorphic respecting the type of variables*, denoted by $E \simeq E'$, if there is a type-respecting bijection $h : \mathsf{Var}(E) \to \mathsf{Var}(E')$ such that $E' = \{h(e) \mid e \in E\}$, where $h(e)$ denotes the computation obtained from e by renaming each variable x with $h(x)$.

For two observable sets O and O' with the same size, it is easy to see that O and O' have the same distribution type if $\{\mathcal{E}(x) \mid x \in O\} \simeq \{\mathcal{E}(x') \mid x' \in O'\}$.

One may notice that constants have to be preserved in the definition of isomorphic with respect to the type of variables. In general, changing a constant in E may change its distribution type. For instance, let us consider a family of sets E_i of (simplified) computations taking from the fourth-order masked implementation of the Sbox [114]:

$$E_{i,j} := \{x_0,\ \mathrm{Sbox}(k \oplus j \oplus x_0) \oplus r,\ \mathrm{Sbox}(k \oplus i \oplus x_0) \oplus r\}, \mathrm{for} 0 \leq i \neq j \leq 255,$$

where x_0 and r are two random variables and k is a private input. In this case, for any distinct pairs of constants (i, j) and (i', j'), $E_{i,j} \simeq E_{i',j'}$ does not hold; thus, we cannot infer the distribution of $E_{i,j}$ from $E_{i',j'}$, although they are almost identical.

To address this issue, we propose a generalization taking into account constants. Our idea is inspired on the observation that some constant can be assimilated without affecting the distribution of computations. For instance, regarding $k \oplus j$ to be k', then $k \oplus i \equiv (k' \oplus j) \oplus i \equiv k' \oplus (j \oplus i)$. Suppose the distribution of $E_{1,2}$ is known and by applying $k \oplus i \equiv k' \oplus (j \oplus i)$, $E_{1,2}$ is normalized as (note $3 = 1 \oplus 2$):

$$\mathtt{norm}(E_{1,2}) := \{x_0,\ \mathrm{Sbox}(k' \oplus x_0) \oplus r,\ \mathrm{Sbox}(k' \oplus 3 \oplus x_0) \oplus r\}.$$

Then, for any $0 \leq i \neq j \leq 255$ such that $(j \oplus i) = 3$, by applying $k \oplus i \equiv k' \oplus (j \oplus i)$, $E_{i,j}$ is also normalized as

$$\mathtt{norm}(E_{i,j}) := \{x_0,\ \mathrm{Sbox}(k' \oplus x_0) \oplus r,\ \mathrm{Sbox}(k' \oplus 3 \oplus x_0) \oplus r\}.$$

We can observe that $E_{i,j} \simeq E_{1,2}$; thus, $E_{i,j}$ has the same distribution as $E_{1,2}$. This idea is formalized in the following definition.

Definition 5.7. A constant c is *assimilable* in a set E of computations if E can be transformed into a set E' of equivalent computations by algebra laws such that all occurrences of the constant c in E' are within the context of $x \circ c$ for operator $\circ \in \{\oplus, +, -\}$ and some variable x, such that x is either not used elsewhere or used as $x \circ c'$ for some constant c' (note that $c \neq c'$).

If c is assimilable in E, we denote by $\mathtt{norm}(E)$ the set of normalized computations that is obtained from E' by iteratively replacing

(1) all occurrences of the constant c in E' (as $x \circ c$) by x, and
(2) every possible $x \circ c'$ (for $c' \neq c$) by $x{\circ}c''$, where $c'' = c' \mathbin{\widehat{\circ}} c$, $\widehat{+} = -$, $\widehat{-} = -$ and $\widehat{\oplus} = \oplus$.

By this replacement, $\mathtt{norm}(E) = E'[x/(x \circ c)][\forall (x \circ c') \textit{ in } E' : (x{\circ}c'')/(x \circ c')]$, one can reduce the number of constants in E.

Example 5.8. Let us consider $e = (x \oplus 1) + (x \oplus 2) + (y \oplus 1)$. The constant 1 is *not* assimilable because there are two occurrences of 1 that are within two different contexts $x \oplus 1$ and $y \oplus 1$, respectively. However, 2 is assimilable (by x), as 2 occurs in the context of $x \oplus 2$ and x occurs elsewhere in $x \oplus 1$. After replacing $x \oplus 2$ by x and $x \oplus 1$ by $x \oplus 3$, we get $(x \oplus 3) + x + (y \oplus 1)$, in which 1 becomes assimilable by y. Finally, $\mathtt{norm}(E) = (x \oplus 3) + x + y$.

Theorem 5.9. *For observable sets $\mathcal{O}$ and $\mathcal{O}'$, if* $\mathtt{norm}(\{\mathcal{E}(x) \mid x \in \mathcal{O}\}) \simeq \mathtt{norm}(\{\mathcal{E}(x) \mid x \in \mathcal{O}'\})$, *then $\mathcal{O}$ and $\mathcal{O}'$ have the same distribution types.*

Proof. Let $E = \{\mathcal{E}(x) \mid x \in \mathcal{O}\}$ and $E' = \{\mathcal{E}(x) \mid x \in \mathcal{O}'\}$. Let n be the number of constants assimilated when computing $\mathtt{norm}(E)$ and $\mathtt{norm}(E')$. We prove by applying induction on n.

- **Base case** $n = 0$. Then, $\mathtt{norm}(E)$ and $\mathtt{norm}(E')$ are isomorphic respecting the type of variables. Let $h : \mathsf{Var}(E) \to \mathsf{Var}(E')$ be the type-respecting bijection; then for every pair $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, there exists a pair $(\eta'_1, \eta'_2) \in \Theta^2_{=X_p}$ such that $\eta_i(x) = \eta'_i(h(x))$ for all $i \in \{1, 2\}$ and $x \in (X_p \cup X_k) \cap \mathsf{Var}(E)$. Moreover, $[\![P]\!]^{\mathcal{O}}_{\eta_1} = [\![P]\!]^{\mathcal{O}}_{\eta_2}$ iff $[\![P]\!]^{\mathcal{O}'}_{\eta'_1} = [\![P]\!]^{\mathcal{O}'}_{\eta'_2}$.
  For every pair $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, there exists a pair $(\eta'_1, \eta'_2) \in \Theta^2_{=X_p}$ such that $\eta_i(h^{-1}(x)) = \eta'_i(x)$ for all $i \in \{1, 2\}$ and $x \in (X_p \cup X_k) \cap \mathsf{Var}(E')$, and $[\![P]\!]^{\mathcal{O}}_{\eta_1} = [\![P]\!]^{\mathcal{O}}_{\eta_2}$ iff $[\![P]\!]^{\mathcal{O}'}_{\eta'_1} = [\![P]\!]^{\mathcal{O}'}_{\eta'_2}$. Thus, the result immediately follows.
- **Inductive set** $n \geq 1$. Without loss of generation, we assume that c is assimilated by x as $x \circ c$ in E, and $x \circ c_1, \ldots, x \circ c_k$ are all the occurrences of x with constants $c_1, \ldots, x_k$. Then, for every $\eta_1 \in \Theta$, $[\![P]\!]^{\mathcal{O}}_{\eta_1}$ using E and $[\![P]\!]^{\mathcal{O}}_{\eta_1[(\eta_1(x) \circ c)/x]}$ using $E[x/(x \circ c)][(x \circ c'_1)/(x \circ c_1), \ldots, (x \circ c'_k)/(x \circ c_k)]$ have the same distribution, where $c'_i = c_i \mathbin{\widehat{\circ}} c$ for all i.
  By symmetry, for every $\eta_2 \in \Theta$, $[\![P]\!]^{\mathcal{O}}_{\eta_2}$ using E and $[\![P]\!]^{\mathcal{O}}_{\eta_2[(\eta_2(x) \circ c)/x]}$ using $E[x/(x \circ c)][(x \circ c'_1)/(x \circ c_1), \ldots, (x \circ c'_k)/(x \circ c_k)]$ have the same distribution.
  Therefore, for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, $[\![P]\!]^{\mathcal{O}}_{\eta_1} = [\![P]\!]^{\mathcal{O}}_{\eta_2}$ using E iff $[\![P]\!]^{\mathcal{O}}_{\eta_1[(\eta_1(x) \circ c)/x]} = [\![P]\!]^{\mathcal{O}}_{\eta_2[(\eta_2(x) \circ c)/x]}$ using $E[x/(x \circ c)][(x \circ c'_1)/(x \circ c_1), \ldots, (x \circ c'_k)/(x \circ c_k)]$.
  By applying the induction hypothesis: for every $(\eta_1, \eta_2) \in \Theta^2_{=X_p}$, there exists a pair $(\eta'_1, \eta'_2) \in \Theta^2_{=X_p}$ such that $\eta_i(h^{-1}(x)) = \eta'_i(x)$ for all $i \in \{1, 2\}$ and $x \in (X_p \cup X_k) \cap \mathsf{Var}(E')$, and $[\![P]\!]^{\mathcal{O}}_{\eta_1} = [\![P]\!]^{\mathcal{O}}_{\eta_2}$ using E iff $[\![P]\!]^{\mathcal{O}'}_{\eta'_1} = [\![P]\!]^{\mathcal{O}'}_{\eta'_2}$. Hence, $[\![P]\!]^{\mathcal{O}}_{\eta_1[(\eta_1(x) \circ c)/x]} = [\![P]\!]^{\mathcal{O}}_{\eta_2[(\eta_2(x) \circ c)/x]}$ using $E[x/(x \circ c)][(x \circ c'_1)/(x \circ c_1), \ldots, (x \circ c'_k)/(x \circ c_k)]$ iff $[\![P]\!]^{\mathcal{O}'}_{\eta'_1} = [\![P]\!]^{\mathcal{O}'}_{\eta'_2}$. Thus, the result immediately follows. □

Note that the pattern-matching-based method could be used to match secure sets and for program debugging. When a new program is just a minor revision of a verified program, this method may be able to quickly check many observable sets.

6 IMPLEMENTATION

We have implemented our methods in the tool HOME. We use Z3 [50] as the underlying SMT solver (fixed size bit-vector theory) for the SMT-based method. The tool works as follows:

(1) Apply Algorithm 1 to compute the set of potential leaky observable sets.
(2) For each procedure call $\mathsf{Check}(\{C_{i,O}\}_{(i,O)\in\mathcal{Y}})$ at Line 14; when the type inference fails to derive any distribution type of $\bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}$, check whether there is a recorded set of computations E' such that $\mathtt{norm}(E) \simeq \mathtt{norm}(E')$ via the pattern-matching-based method, where E is the set of computations $\{\mathcal{E}(x) \mid x \bigcup_{(i,O)\in\mathcal{Y}} C_{i,O}\}$ after transformations (e.g., $\mathsf{Simply}_{\mathsf{Alg}}$, $\mathsf{Simply}_{\mathsf{Dom}}$ and $\mathsf{Simply}_{\mathsf{Col}}$); if E' exists, then return the distribution type of E'.
(3) If E' does not exist, apply model-counting methods to the set E and record E with its corresponding distribution type for later pattern matching.

Finally, PLS contains exactly the set of leaky observable sets. Note that we do not apply pattern-matching- and model-counting-based methods to observable sets whose size is greater than the security order d for efficiency considerations.

To boost the performance of the model-counting method, we implement a GPU-accelerated parallel algorithm, as described below.

6.1 GPU-Accelerated Parallel Algorithm

In this subsection, we show how to leverage GPU's superior compute capability to check satisfiability of Ω^O in Equation (1). In general, given a potential leaky observable set O, we automatically synthesize a GPU program from the computations of observable variables in O such that the GPU program outputs SAT iff Ω^O is satisfiable, i.e., the program is O-leaky.

Our work is based on CUDA, a parallel computing platform and programming model for NVIDIA GPUs. Specifically, we utilize Nvidia GeForce GTX 1080 (Pascal) with compute capability 6.1. From a programming perspective, the CUDA architecture defines three levels of threads, i.e., grid, block, and warp, to organize units. A warp consists of 32 consecutive threads that are executed in the Single Instruction Multiple Thread fashion on Streaming Processors; namely, all threads execute the same instruction, and each thread carries out that operation on its own private data. A block running on Streaming Multiprocessors contains at most 32 wraps (giving rise to 32×32 threads). The maximum number of blocks in a grid is $65,535 \times 65,535$, and each grid runs on the Scalable Streaming Processor Array. The code running on GPUs is usually referred to as *Kernel*.

We parallelize Algorithm 2 as a CUDA program. In this work, we illustrate the idea on byte programs; i.e., each variable is of 8-bit. Typically, the number of random variables is usually much larger than that of the other variables. Therefore, we enumerate assignments of random variables in GPUs while enumerating valuations of public and input variables in CPUs. Namely, the Counting function in Algorithm 2 is implemented as a Kernel. However, it would be difficult to implement a generic Kernel to compute distributions of sets of computations $(\mathcal{E}(x))_{x\in O}$, unless computations are evaluated by traversing their abstract syntax trees, which is control-flow intensive and would downgrade the GPU performance. As a result, instead of designing a generic Kernel, for each observable set O, we automatically synthesize a CUDA program that checks whether Ω^O is satisfiable based on Algorithm 2.

ALGORITHM 3: The skeleton of synthesized GPU programs

```
__device__ unsigned char OP1(...)
    ...;
...
__device__ unsigned char OPJ(...)
    ...;
__device__ unsigned char EXP_1(η_p, η_k, η_r, threadIdx, blockIdx)
    ...;
...
__device__ unsigned char EXP_m(η_p, η_k, η_r, threadIdx, blockIdx)
    ...;
int main GPUBFEnum(P, O = {x_1, ..., x_m})
    int *D_1; int *D_2;
    cudaMallocManaged(&D_1, 256^m);
    cudaMallocManaged(&D_2, 256^m);
    dim3 block(16,16);
    dim3 grid(4096/block.x,4096/block.y);
    forall η_p : ⋃_{x∈O} X_p ∩ Var(E(x)) → I do
        memset(D_1, 0, sizeof(unsignedchar));
        b := false;
        forall η_k : ⋃_{x∈O} X_k ∩ Var(E(x)) → I do
            memset(D_2, 0, sizeof(unsignedchar));
            if b = false then
                KernelCounting<<<grid, block>>> (D_1, η_p, η_k, O);
                cudaDeviceSynchronize();
                b := true;
            else
                KernelCounting<<<grid, block>>> (D_2, η_p, η_k, O);
                cudaDeviceSynchronize();
                if D_1 ≠ D_2 then
                    return SAT;
    return UNSAT;
__global__ void KernelCounting(D, η_p, η_k, O)
   {r_1, ..., r_h} := ⋃_{x∈O} RVar(E(x));
   forall η_r : {r_4, ..., r_h} → I do
       c_1 :=EXP_1(η_p, η_k, η_r, threadIdx, blockIdx);
       ...
       c_m :=EXP_m(η_p, η_k, η_r, threadIdx, blockIdx);        // m = |O|
       index := Σ_{i=0}^{m-1} c_i × 256^i;
       atomicAdd(&D[index], 1);
```

The numbers of threads per block and blocks per grid in each synthesized CUDA program are determined by the number $R := |\bigcup_{x\in O} \mathsf{RVar}(\mathcal{E}(x))|$ of random variables in $(\mathcal{E}(x))_{x\in O}$. If $R = 3$, we choose 2-D $(16, 16)$ blocks each of which has 2^8 threads, and 2-D $(256, 256)$ grids each of which has 2^{16} blocks. (Note that the number 2^{24} of threads exactly corresponds to the number of valuations of three 8-bit random variables.) Moreover, we do not need to enumerate those valuations, as the thread Id and block Id (i.e., threadIdx and blockIdx in CUDA) of each host thread in GPU exactly correspond one of those valuations. If $R < 3$, we reduce the number of blocks and/or threads such that the total number of threads is the number of valuations of random variables. Otherwise, $R > 3$, and we set 2^8 number of threads in each block and 2^{16} number blocks in one grid for three random variables, while the valuations of the rest of the random variables are enumerated in GPU.

For each operation used in computations of $(\mathcal{E}(x))_{x \in O}$ but not supported in CUDA, we synthesize a corresponding __device__ function, which will be called from GPUs only and executed therein. For each computation $\mathcal{E}(x)$, we also synthesize a __device__ function, which computes the value of $\mathcal{E}(x)$ using __device__ functions for operations based on thread Id and block Id of the host thread, which represent the valuations of some random variables.

For memory management, we use int arrays to store distributions, which are accessed from both CPU for comparing distributions (read-only) and GPU for computing distributions (read and write). We utilize unified memory provided by CUDA to allocate memory for both int arrays, namely, to allocate memory by invoking the cudaMallocManaged function, by which the managed pointers to int arrays are valid on both the GPU and CPU. To resolve data race, the update of int arrays in Kernel is performed in one atomic transaction (via the atomicAdd function in CUDA).

Concretely, Algorithm 3 shows a skeleton of synthesized GPU programs, where the number of random variables is greater than three. Other cases are similar. The __device__ functions implement all the CUDA non-supported operations and expressions that are invoked and executed on GPU. D_1 and D_2 are int arrays for storing distributions. The function KernelCounting is the Kernel that computes distributions for each valuation of public and private input variables. The function KernelCounting is invoked at Line 20 and Line 24 for each valuation of public and private input variables. After each invoking of KernelCounting, the function cudaDeviceSynchronize is invoked, which waits until all preceding commands in all streams of all host threads have completed. In the body of KernelCounting, the valuations of the first three random variables are implicitly represented by threadIdx and blockIdx, while η_r denotes a valuation of other random variables. Finally, the values of expressions are iteratively computed via calling the corresponding __device__ functions. The value vector of expressions is encoded as an index to the array D, where the value at this index increases by 1 atomically to avoid data race.

7 EVALUATION

The experiments were conducted on arithmetic programs over the byte domain. We used a server with 64-bit Ubuntu 16.04.4 LTS, Intel Xeon CPU E5-2690v4, 2.6GHz, and 256GB RAM (only one core is used in our computation). For GPU-based algorithms, we use NVIDIA GeForce GTX 1080 with compute capability 6.1, as mentioned in Section 6.1.

7.1 Evaluation on Higher-Order Masking

We evaluate our methods on implementations of masked arithmetic algorithms, ranging from multiplication algorithms to (round-reduced or full) AES/MAC-Keccak. Some of them are provided by the authors of [8], while the others are implemented according to the published masked algorithms.

The results of our type inference (Algorithm 1 without applying model-counting- and pattern-matching-based methods) are presented in Table 1. Column 1 shows the reference and description of the program, where A2B and B2A denote the implementations of conversion algorithms from Boolean-to-arithmetic masking and arithmetic-to-Boolean masking, respectively, SecH and SecR denote the implementations of the non-linear transformation and the round function of Simon [117], and DOM AND is a $GF(2^8)$ version from [69]. Columns 2 to 7 show the statistics of Algorithm 1, including the numbers of potential leaky observable sets, tuples that should be considered with respect to masking order d (i.e., all non-empty subsets of X_o with size $\leq d$) in order to

Table 1. Experimental Results of Type Inference on Masked Programs

Description	HOME						[8]			
	Result	**♯Tuples**	**♯Sets**	♯Simply$_{\text{Dom}}$	♯Simply$_{\text{Col}}$	**Time (s)**	**Result**	**♯Tuples**	**♯Sets**	**Time (s)**
First-Order Masking										
Multiplication [111]	0	13	6	5	0	≈0	0	13	7	≈0
Sbox (4) [48]	0	73	15	14	0	≈0	0	64	17	≈0
Full AES (4) [48]	0	20,060	**515**	514	0	**2**	0	17,206	3,342	128
Full Keccak [8]	0	18,218	**2,813**	2,813	0	**83**	0	13,466	5,421	405
B2A [64]	**1**	10	4	2	0	≈0	**N/A**			
A2B [64]	**37**	48	39	1	0	0.15	**N/A**			
A2B [46]	0	1,448	14	13	0	≈0	**N/A**			
B2A [46]	0	2,494	2	1	0	≈0	**N/A**			
A2B [45]	**45**	86	56	12	0	≈0	**N/A**			
B2A [21]	0	19	3	2	0	≈0	**N/A**			
B2A [42]	**1**	14	4	2	0	≈0	**N/A**			
Second-Order Masking										
Sbox [114]	0	1,188,111	**1,285**	1,284	256	**1.073**	0	1,188,111	4,104	1.649
Multiplication [111]	0	435	52	51	0	0.001	0	435	92	0.001
Sbox [111]	**2**	7,503	**270**	267	0	0.05	2	7,140	866	0.045
Key schedule [111]	0	31,828,231	**475,943**	475,942	0	**3,087**	0	23,041,866	771,263	340,745
B2A [21]	0	1,653	25	23	0	≈0	**N/A**			
B2A [113]	0	780	15	13	0	≈0	**N/A**			
SecH (2) [117]	0	1,770	14	13	0	≈0	**N/A**			
SecR [117]	0	3,003	25	24	0	≈0	**N/A**			
DOM AND [69]	0	435	46	45	0	≈0	**N/A**			
Third-Order Masking										
Multiplication [111]	0	24,804	713	712	0	**0.021**	0	24,804	1,410	0.033
Sbox (4) [48]	0	6,784,540	**18,734**	18,733	0	**2.021**	0	4,499,950	33,075	3.894
Sbox (5) [48]	0	6,209,895	**10,470**	10,469	0	**3.757**	0	4,499,950	39,613	5.036
B2A [46]	0	274,884,292,760	7	6	0	0.11	**N/A**			
B2A [21]	0	457,310	816	807	0	0.052	**N/A**			
B2A [113]	0	59,640	133	132	0	≈0	**N/A**			
DOM AND [69]	0	23,426	572	571	0	≈0	**N/A**			
Fourth-Order Masking										
Sbox [114]	**98,176**	4,874,429,560	**1,087,630**	924,173	821,888	**702**	98,176	4,874,429,560	35,895,437	22,119
Multiplication [111]	0	2,024,785	12,845	12,844	0	**0.534**	0	2,024,785	33,322	1.138
Sbox (4) [48]	0	3,910,710,930	**1,159,295**	1,159,294	0	**376**	0	2,277,036,685	3,343,587	879
B2A [21]	0	387,278,970	62,570	62,561	0	10.7	**N/A**			
B2A [113]	0	6,438,740	1,271	1,270	0	0.11	**N/A**			
DOM AND [69]	0	2,024,785	10,626	10,625	0	0.71	**N/A**			
Fifth-Order Masking										
Multiplication [111]	0	216,071,394	**281,731**	281,730	0	**15**	0	216,071,394	856,147	45
Sbox (4) [48]	0	2,782,230,535,161	99,996,680	99,996,679	0	49,598	**N/A**			
B2A [113]	0	901,289,592	29,926	29,838	0	3.03	**N/A**			

compare with the tool of [8], sets actually checked by Algorithm 1, sets whose verification involves the Simply$_{\text{Dom}}$ and Simply$_{\text{Col}}$ transformations, and verification time (*excluding* program parsing). Likewise, Columns 8 to 11 show the results reported in [8], the unique sound (but incomplete) approach that is able to automatically verify masked implementations of higher-order arithmetic programs under an equivalent leakage model of the ISW model. Since the tool of [8] is unavailable, in Columns 8 to 11, we simply replicate the statistics of the BBDFGS algorithm from the paper [8] when it is available (N/A is marked otherwise). Recall that [8] used a different experimental setup: a headless VM with a dual-core 64-bit processor clocked at 2GHz (only one core is used in the computation). Note that Sbox [114] under fourth-order masking is verified under third-order security only in order to compare with [8], while other benchmarks are verified under their masking orders.

Results on common benchmarks. All the programs not marked as N/A in Columns 8 to 11 are provided by the authors of [8]. We only did necessary pre-processing, e.g., transformed them into SSA form. Because of this, from Columns 3 and 9 (i.e., ♯Tuples), one can see that we considered more tuples in several benchmarks (e.g., Full AES (4) [48], Full MAC-Keccak [8], Sbox [111], Key schedule [111], Sbox (4) [48], Sbox (5) [48]) than [8]; namely, we considered more observable variables than [8].

From the experimental results, we can observe that there are two benchmarks (i.e., Sbox [111] under second-order masking and Sbox [114] under fourth-order masking) that have potential leaky observable sets, and Algorithm 1 produces the same number as [8]. This demonstrates that Algorithm 1 is at least as precise as the one in [8]. We will report in Section 7.2 the results of resolving these potential leaky observable sets using our model-counting and pattern-matching methods.

From Columns 4 and 10 (i.e., ♯Sets), one can observe that the number of observable sets actually verified by Algorithm 1 is less than the one in [8] on all the common benchmarks (despite there being more observable variables to be considered in several benchmarks). The differences are noticeable on several benchmarks (e.g., Full AES (4), Full Keccak, Sbox, Multiplication, Sbox [48, 111], and Key schedule). Reducing the number of verified observable sets allows us to verify fifth-order Sbox (4) [48], which has not been done in [8]. Furthermore, from Columns 7 and 11 (i.e., Time), we observe that Algorithm 1 is faster than [8] on almost all the benchmarks, and the improvement is significant on larger benchmarks (e.g., 110X, 64X, and 31X speed-up for Key schedule, Full AES (4), and fourth-order Sbox [114]). These results demonstrate the performance of our type inference algorithm. Furthermore, the algorithm presented in [8] has an issue that may miss the verification of some observable sets. (We have informed some authors of [8].)

Results on new benchmarks. All the programs marked as N/A are new benchmarks. We note that B2A [42] in Common Lisp has been *semi-automatically* verified under the ISW model by Coron [43]; the AES implementation [48] including Sbox (4) has been *semi-automatically* proved under the d-NI model [9]. Some of the first-order A2B and B2A (except A2B [45]) have been verified in [60]. All the other higher-order benchmarks have not been verified by computer-aided tools.

From Table 1, we can observe that almost all benchmarks can be proved secure using our type inference algorithms in a few seconds. The exceptions include B2A [64], A2B [64], A2B [45], and B2A [42] which respectively have 1, 37, 45, and 1 potential leaky observable set(s). We shall see in Section 7.2 that these potential leaky observable sets are actually spurious using model counting. To our knowledge, it is the first time that these higher-order programs are *automatically* proved secure by computer-aided tools. Recall that A2B and B2A are two kinds of conversion algorithms between arithmetic and Boolean masking. Our tool could be used to verify masked implementations of cryptographic algorithms that use A2B and/or B2A conversion algorithms.

Usage of the transformations $\mathsf{Simply}_{\mathsf{Dom}}$ and $\mathsf{Simply}_{\mathsf{Col}}$. Columns 5 and 6 show the number of sets whose verification involves the transformations $\mathsf{Simply}_{\mathsf{Dom}}$ and $\mathsf{Simply}_{\mathsf{Col}}$, respectively. We can see that $\mathsf{Simply}_{\mathsf{Dom}}$ is heavily used, while $\mathsf{Simply}_{\mathsf{Col}}$ is used only in one benchmark (i.e., second-order and fourth-order Sbox [114]), which allows to prove lots of observable sets (e.g., 256 on second-order Sbox [114]) without invoking model counting. Moreover, $\mathsf{Simply}_{\mathsf{Dom}}$ and $\mathsf{Simply}_{\mathsf{Col}}$ simplify the expressions of the 98,176 potential leaky observable sets for Sbox [114], so that

Table 2. Comparison of Three Model-Counting Methods

Description	Order	♯CNT	Result	SMT	BFEnum	GPU
k^3 [111]	1	2	2	96m	**0.2s**	0.43s
k^{254} [111]	1	4	4	O.T.	30m	**7.03s**
B2A [64]	1	1	0	17s	2s	**0.86s**
A2B [64]	1	37	0	O.T.	O.T.	**33.18s**
A2B [45]	1	45	0	O.T.	O.T.	**160m**
B2A [42]	1	1	0	1m 35s	10m 59s	**3.17s**
Sbox [111]	2	2	1(1)	O.T.	O.T.	**3,600s**
Sbox [114]	4	766	98,176	O.T.	O.T.	**323s**

O.T. denotes run out of time (3 hours).

pattern-matching and model-counting methods can be easily applied. Note that statistics of the transformation $\text{Simply}_{\text{Alg}}$ are not reported, as its complexity is of constant time and is negligible.

Note that our experimental setting is better than the one of [8]. We also conducted experiments on a server with Intel(R) Xeon(R) CPU E5-2603v4@1.70GHz (only one core is used in the computation) and 32G RAM. The time cost increased slightly (e.g., the time on Full AES and Full Keccak becomes 3.4s and 134.3s, respectively), but is still lower than that of [8]. We leave the comparison of the two tools on the same platform as future work.

7.2 Comparison of Model-Counting Methods

One important component of our approach is the model-counting method on which we rely to resolve potential leaky observable sets. As mentioned in Section 1, we consider two baseline algorithms (based on SMT encoding and brute-force methods) and a novel GPU-accelerated parallel algorithm. For the sake of evaluation, we carry out experiments only on programs that have potential leaky observable sets reported by our type inference algorithm (cf. Result in Table 1). We also implemented two programs for computing k^3 and k^{254}, which contain one private input variable and three and five random input variables, respectively. These programs are taken from the first-order secure exponentiation [111] without the first RefreshMask function.

Table 2 shows the statistics of the three model-counting methods, with time limited to 3 hours per program. Column 1 shows the reference and description of the program. Column 2 shows the security order. Column 3 (♯CNTs) shows the time of the model-counting method. Column 4 shows the number of genuine leaky observable sets. Columns 5 to 7 show the verification time (*excluding* the time for type inference algorithm) of the SMT-based, (naïve) brute-force, and GPU-accelerated parallel methods, respectively.

The resolution shows that all potential leaky observable sets of B2A [64], A2B [64], A2B [45], and B2A [42] are spurious, while all potential leaky observable sets of Sbox [114] are genuine. On program Sbox [111], we resolved one of two potential leaky observable sets as a genuine one in 1 hour, but the other set cannot be resolved in 2 hours, which is the only case that was unsuccessful in our experiments.

In detail, the GPU-accelerated parallel method significantly outperforms the other two methods on large programs. In particular, the SMT-based and brute-force methods run out of time on five and four programs, respectively. On the small program k^3, the brute-force method is significantly faster than the SMT-based one, and is also faster than the GPU-accelerated one. The latter is because the GPU-accelerated method synthesizes a GPU program for each expression and the involved I/O cost is remarkable in small programs. The GPU-accelerated algorithm provides two orders of magnitude improvements on the program k^{254}. A2B [64] has been verified in [60] based on the oracle provided by the authors. However, it is not always the case that one can find such an oracle luckily. It runs out of time if we use the SMT-based method or the brute-force method without the oracle, while the GPU-accelerated method can verify this program in less than 1 minute. As a conclusion, when model counting is concerned, we recommend the GPU-accelerated algorithm.

For the fourth-order Sbox [114], which is faulty, it only took 4 minutes to automatically resolve all of the 98,176 potential leakage sets as genuine ones. Therefore, our tool is still faster than [8], albeit it needs to invoke the model-counting method to resolve those sets that cannot be determined by type inference. It should be emphasized that this was not possible without the pattern-matching-based method described in Section 5.3. Indeed, we estimate (based on the experiment) that each set takes approximately 0.5s, and in total they would require approximately 14 hours. Instead, we identified 766 ($255 \times 2 + 256$) patterns that can be used to handle all 98,176 potential leaky observable sets. As a result, only 766 times of model counting are needed, which took less than 7 minutes, i.e., two orders of magnitude faster.

The 766 patterns are summarized as follows:

(1) $\{x_0,\ \mathrm{Sbox}(k \oplus x_0) \oplus r,\ \mathrm{Sbox}(k \oplus i \oplus x_0) \oplus r\}$;
(2) $\{\mathrm{Sbox}(k) \oplus r,\ \mathrm{Sbox}(k \oplus x_0) \oplus r,\ \mathrm{Sbox}(k \oplus i \oplus x_0) \oplus r\}$;
(3) $\{x_0,\ \mathrm{Sbox}(k) \oplus r,\ \mathrm{Sbox}(k \oplus x_0 \oplus j) \oplus r\}$;

where $0 < i \leq 255$ and $0 \leq j \leq 255$, x_0 is a random variable, k is a private input, and r is a random variable that is introduced by our transformations. The family in Item (1) captures 65,280 observable sets, namely, 256 observable sets for each $0 < i \leq 255$; the family in Item (2) captures 32,640 observable sets, namely, 128 observable sets for each $0 < i \leq 255$; and the family in Item (3) captures 256 observable sets, 1 observable set for each $0 \leq j \leq 255$.

Barthe et al. [8] manually analyzed the 98,176 potential leaky observable sets that are summarized by four families. These are similar to our automatically computed patterns except for the patterns in Item (3), which is $\{x_0,\ y_0,\ \mathrm{Sbox}(k \oplus x_0 \oplus j) \oplus r\}$ with $y_0 = \mathrm{Sbox}(x_0)$ in [8] (note the third expression is adjusted for the sake of presentation). After manually analyzing source code of Sbox [114] under fourth-order masking, we confirm that our pattern is correct, while the pattern in [8] is not correct. This demonstrates that it is hard to manually examine potential leaky observable sets.

7.3 Comparison with `maskVerif`

Our tool HOME is designed to tackle arithmetic programs, but it is also interesting to evaluate its performance on Boolean programs, for which we compare with the latest version of the open source tool `maskVerif` [10], which is limited to Boolean programs. To the best of our knowledge, `maskVerif` is the only open source tool for verifying higher-order Boolean programs. We experiment on the largest six Boolean programs (P12 to P17) from [54], which are one-round versions of the full 24-round MAC-Keccak [8], together with randomly selected benchmarks from `maskVerif`.

In our experiment, `maskVerif` reported "stack overflow" error on P12 to P17. (We have reported this issue to the developers of `maskVerif`.) For the sake of experiments, we removed the last 5,000 assignments for each program when testing `maskVerif`, while our tool HOME is still tested on the whole programs P12 to P17. (For the abridged version no "stack overflow" error was reported from `maskVerif`.) We also revised DOM AND [69] and DOM Keccak Sbox [70] by introducing the following extra dummy variables and statements:

$$t_1 = r_1 \wedge x;\ t_2 = (\neg r_1) \wedge (\neg x);\ t_3 = t_1 \wedge t_2;\ t_4 = t_2 \wedge r_3;\ \ldots\ t_{18} = t_{16} \wedge r_{17};\ t_{19} = t_{17} \wedge r_{18};$$

where r_1 to r_{17} are fresh random variables, and x denotes a share of a private input variable. Obviously, t_3 to t_{19} are always 0.

Table 3. Comparison with maskVerif

Description	Time (s)		Result
	maskVerif [10]	HOME	
First-Order Masking			
DOM AND [69]	0.01	0.01	0
DOM Keccak Sbox [70]	0.01	0.01	0
DOM AES Sbox [69]	**0.23**	4.52	0
TI Fides-192 APN [23]	**86.61**	139.40	0
P12 [54]	3,223	**2.9**	0
P13 [54]	3,257 (1,234)	**122**	4.8k
P14-P17 [54]	O.T. (≤12)	**72-168**	1.6k-17.6k
Second-Order Masking			
DOM AND [69]	0.01	0.01	0
DOM AND (Revised) [69]	16.82	**0.89**	0
DOM Keccak Sbox [70]	**0.01**	0.05	0
DOM Keccak Sbox(Revised) [70]	16.62	**1.93**	0
DOM AES Sbox [67]	**61.59**	7,385	0
Third-Order Masking			
DOM AND [69]	**0.01**	0.02	0
DOM AND(Revised) [69]	828.70	**6.39**	0
DOM Keccak Sbox [70]	**0.33**	1.26	0
DOM Keccak Sbox(Revised) [70]	1,041.67	**27.40**	0
Fourth-Order Masking			
DOM AND [69]	**0.13**	0.40	0
DOM AND(Revised) [69]	O.T.	**77.40**	0
DOM Keccak Sbox [70]	**16.35**	78.13	0
DOM Keccak Sbox(Revised) [70]	O.T.	**690.79**	0

Table 3 presents the results, with time being limited to 2 hours per program. Column 1 gives the programs under comparison. Columns 2 and 3 show the verification time of maskVerif and our tool, respectively. Column 4 gives the number of leaky observable sets. On the programs taken from maskVerif, maskVerif performs better (up to 5×) than HOME. We note that there is one benchmark (second-order) DOM AES Sbox for which maskVerif performs exceptionally well. The major reason is that an ad hoc rule is used therein but could not be used in HOME because it is tailored for Boolean programs. It is perhaps worth pointing out that we have identified some bugs of maskVerif. For instance, when maskVerif verifies DOM AND (under second order), the leaky observable set $\{(k \oplus r_0 \oplus r_1) \wedge r_2, r_0, r_1, r_2\}$ where k is private and r_0, r_1, r_2 are random variables is considered secure. This bug has inadvertently reduced the verification time of maskVerif as fewer sets of variables need to be examined. (We have reported this issue to the developers of maskVerif.)

On the programs P12 to P17 and revised programs DOM AND [69] and DOM Keccak Sbox [70], HOME significantly outperforms maskVerif. Specifically, on the secure program P12, HOME takes 2.9s, while maskVerif takes 3,223s on the reduced version. On the insecure program P13, HOME identified all the flaws of the program in 122s, while maskVerif identified 1,234 flaws of the reduced version in 3,257s. On the insecure programs P14 to P17, HOME identified all the flaws using at most 168s, while maskVerif runs out of time (2 hours) and identified at most 12 flaws. On second-/third-order revised programs, HOME is 8.6 to 130× faster than maskVerif. On fourth-order revised programs, maskVerif ran out of time.

In conclusion, even for Boolean programs, HOME demonstrates largely comparable performance on the benchmarks tested by maskVerif, and indeed considerably better performance on the new benchmarks.

8 RELATED WORK

In this section, we review related work on masking countermeasures in general, as well as existing techniques on the analysis of masked programs and the detection/mitigation of other types of side-channel leaks.

Masking. Boolean and arithmetic masking schemes [19, 26, 31, 57, 64, 75, 92, 93, 100, 107, 110, 111, 114, 122] have been widely investigated in the past two decades with differences in adversary models, masking schemes, cryptographic algorithms, and compactness. Secure conversion algorithms between Boolean and arithmetic maskings have also been investigated [21, 42, 45, 46, 64, 74, 113]. These countermeasures and conversion algorithms are often designed manually for specific cryptographic algorithms. In this context, the common problem is the lack of efficient and effective tools for automatically proving their correctness [47, 48]. Our work aims to bridge this gap.

Testing. The predominant approach addressing the security of (masked) implementations of cryptographic algorithms is the empirical leakage assessment by statistical significance tests or launching state-of-the-art side-channel attacks [15, 16, 36, 49, 63, 72, 80, 87, 88, 91, 96, 97, 103, 106, 124]. These approaches are valuable in identifying flaws even without any knowledge of the leakage model, but can neither prove their absence nor identify all flaws, due to the limitation in measurement setup and/or explored traces. This article purses an alternative, formal verification-based approach that is largely complementary to the work based on testing.

Formal verification. Formal verification approaches, which are able to prove the absence of side-channel leaks, have been proposed in prior work [8–10, 18, 20, 24, 25, 27, 43, 54, 55, 60, 61, 94, 101, 120, 129]. However, as we have explained earlier, these existing formal verification methods are limited in applicability (i.e., Boolean program, stronger leakage model, or first-order security only) and accuracy (i.e., false alarms).

Early work via type-based proof systems refers to [18, 94], which checks if a computation result is logically dependent of the secret data and, at the same time, logically independent of any random variable used for masking the secret data. However, these incomplete approaches only support verification of first-order arithmetic programs and may even be unsound under the ISW model, as pointed out in [54].

To improve accuracy, Eldib et al. proposed a model-counting-based method [54, 55] that is both sound and complete under the ISW model. This method reduces the verification problem to a series of satisfiability problems encoding model-counting constraints, which is solved by leveraging SMT solvers. However, it is limited to the first-order Boolean programs only. Blot et al. extended this SMT-based method to verify higher-order programs [27]. The SMT encoding is exponential in the number of bits of random variables and the number of orders, and hence is short of scalability and limited to Boolean programs only. Our SMT-based method can be seen as a generalization of these methods. Nevertheless, our GPU-accelerated parallel algorithm significantly outperforms the SMT-based method.

To improve efficiency, Barthe et al. introduced the notion of d-NI to characterize the security of masked programs and proposed a sound proof system to verify higher-order masked programs [8]. The d-NI notion was later extended to d-SNI [9], which enables compositional verification.

However, these approaches are incomplete; namely, it may produce spurious leaky observable sets. Furthermore, as mentioned in Section 7.1, these approaches may miss the verification of some observable sets. In this direction, Bisi et al. [24] proposed a technique for verifying higher-order masking, which was limited to Boolean programs with linear operations only. Ouahma et al. generalized the approach of [8] to verify assembly-level code [101], but it is incomplete and limited to first-order programs only. Coron [43] proposed two complementary semi-automatic approaches via elementary circuit transforms and showed how to generate security proofs automatically, for simple circuits, but they are also incomplete. Barthe et al. developed the unified framework `maskVerif` [10] for both software and hardware implementations, taking into account glitch and transitions, but it is limited to Boolean programs only and their tool missed the verification of some observable sets in our experiments.

As a matter of fact, the most efficient masked programs do not achieve d-SNI directly, as mentioned by Bloem et al. [25]. Thus, Bloem et al. proposed a sound approach [25] via Fourier analysis, which considers the Fourier expansion of the Boolean functions and reduces the verification to checking whether certain coefficients of the Fourier expansion are zero or not [25]. They studied the security problem of Boolean programs/hardware circuits in the d-threshold probing model [75] and its extension with glitches for any given d. The verification problem is solved by leveraging SAT solvers. However, they considered Boolean programs/hardware circuits only. Furthermore, it was shown by Barthe et al. [10] that `maskVerif` outperforms [25]. Belaïd et al. proposed another compositional verification approach in [20] to overcome the limitation of d-SNI [9], but it can only verify Boolean programs composed of ISW multiplication functions, sharewise addition functions, and d-SNI refresh functions.

In our prior work [59–61, 129], we have proposed gradual refinement-based approaches for verifying masked Boolean and arithmetic programs, respectively, which integrate the semantic type system and model-counting-based methods, hence bringing the best of both worlds. This semantic type system was leveraged by Wang et al. [120] to identify transition-based flaws. All these approaches are limited to first-order security only. It is challenging to generalize these approaches to higher-order masked arithmetic programs, which is addressed by the current work.

Compared to the above existing formal verification approaches, the current work studies formal verification of arithmetic programs against a d-threshold probing model for any given d. Both our type system and model-counting-based method significantly improve the applicability and efficiency. Our pattern-matching-based method is novel and effective at reducing the cost of model counting and summarizing patterns of leaky observable sets, which can be used for diagnosis and debugging. Putting them together, our hybrid formal verification approach goes significantly beyond the state of the art in terms of applicability, accuracy, and efficiency.

Automated mitigation of power side-channel flaws. Automated mitigation techniques have been proposed to repair power side-channel flaws [1, 9, 17, 27, 53, 94, 119, 120]. For example, techniques proposed in [1, 9, 17, 94] rely on compiler-like pattern matching, whereas the ones proposed in [27, 53, 119] use inductive program synthesis, and the one in [120] constrains register allocation. All these works either rely upon existing formal verification techniques, and hence have similar limitations as described above, or do not use formal verification techniques, and thus correctness cannot be guaranteed. It would be interesting to investigate whether our new approach can aid in the mitigation of power side-channel flaws, effectively making countermeasures better, as done in [27, 53].

Other types of side channels. In addition to power side-channel attacks, there are other types of side-channel attacks against cryptographic programs, where the side channels can be in the form of, e.g., CPU time, faults, and cache behaviors. Techniques for verification and mitigation of these types of side-channel attacks have been studied in the literature, such as [3, 4, 7, 30, 37, 81, 104, 105, 125, 126] for timing side-channel attacks, [13, 14, 34, 35, 38, 51, 66, 71, 83, 116, 121, 125] for cache side-channel attacks, and [12, 22, 28, 29, 56, 73] for fault attacks. Each type of side-channel has unique characteristics, which usually requires specific verification techniques, so these results are orthogonal to our work.

9 CONCLUSION

In this work, we have proposed a hybrid formal verification approach for higher-order masked arithmetic programs. The approach contains a sound proof system equipped with an efficient algorithm for type inference, which significantly outperforms the approach [8] for arithmetic programs, as well as novel model-counting- and pattern-matching-based methods for resolving potential leaky observable sets automatically that cannot be accomplished by the existing tools. Experimental results show that our approach is not only significantly faster but also applicable to more cryptographic implementations that could not be proved secure automatically before.

Future work includes extending our methods to verifying programs with inherent branching and loops, and/or under other leakage models such as d-NI/SNI or d-threshold probing model, as well as their extensions with glitches and transitions, as done in [10, 25].

ACKNOWLEDGMENTS

The authors would like to thank Professor Gilles Barthe, Professor Benjamin Grégoire, and Professor Chao Wang for providing benchmarks, and the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*. 77–82.

[2] Rasim M. Alguliyev, Yadigar Imamverdiyev, and Lyudmila V. Sukhostat. 2018. Cyber-physical systems and their security issues. *Computers in Industry* 100 (2018), 212–223.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Security Symposium*. 53–70.

[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 362–375.

[5] Victor Arribas, Svetla Nikova, and Vincent Rijmen. 2018. VerMI: Verification tool for masked implementations. In *Proceedings of the 25th IEEE International Conference on Electronics, Circuits and Systems*. 381–384.

[6] Yosef Ashibani and Qusay H. Mahmoud. 2017. Cyber physical systems security: Analysis, challenges and solutions. *Comput. Secur.* 68 (2017), 81–97.

[7] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 193–204.

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

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

[10] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. 2019. maskVerif: Automated verification of higher-order masking in presence of physical defaults. In *Proceedings of the 24th European Symposium on Research in Computer Security*. 300–318.

[11] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. 2017. Parallel implementations of masking schemes and the bounded moment leakage model. In *Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 535–566.

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

[13] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage resilience against concurrent cache attacks. In *Proceedings of the 3rd International Conference on Principles of Security and Trust*. 140–158.

[14] Tiyash Basu, Kartik Aggarwal, Chundong Wang, and Sudipta Chattopadhyay. 2020. An exploration of effective fuzzing for side-channel cache leakage. *Software Testing, Verification & Reliability* 30, 1 (2020).

[15] Lejla Batina, Lukasz Chmielewski, Louiza Papachristodoulou, Peter Schwabe, and Michael Tunstall. 2019. Online template attacks. *Journal of Cryptographic Engineering* 9, 1 (2019), 21–36.

[16] Lejla Batina, Jip Hogenboom, and Jasper G. J. van Woudenberg. 2012. Getting more from PCA: First results of using principal component analysis for extensive power analysis. In *Proceedings of the RSA Conference Cryptographers' Track*. 383–397.

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

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

[19] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. 2017. Private multiplication over finite fields. In *Proceedings of the 37th Annual International Cryptology Conference*. 397–426.

[20] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. 2018. Tight private circuits: Achieving probing security with the least refreshing. In *Proceedings of the 24th International Conference on the Theory and Application of Cryptology and Information Security*. 343–372.

[21] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. 2018. Improved high-order conversion from Boolean to arithmetic masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2 (2018), 22–45.

[22] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference*. 513–525.

[23] Begül Bilgin, Andrey Bogdanov, Miroslav Knezevic, Florian Mendel, and Qingju Wang. 2013. Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In *Proceedings of the 15th International Workshop on Cryptographic Hardware and Embedded Systems*. 142–158.

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

[25] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. 2018. Formal verification of masked hardware implementations in the presence of glitches. In *Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 321–353.

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

[27] Arthur Blot, Masaki Yamamoto, and Tachio Terauchi. 2017. Compositional synthesis of leakage resilient programs. In *Proceedings of the 6th International Conference on Principles of Security and Trust*. 277–297.

[28] Jakub Breier, Xiaolu Hou, and Yang Liu. 2018. Fault attacks made easy: Differential fault analysis automation on assembly code. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 2 (2018), 96–122.

[29] J. Breier, X. Hou, and Y. Liu. 2019. On evaluating fault resilient encoding schemes in software. *IEEE Transactions on Dependable and Secure Computing* (2019), 1–14.

[30] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–37.

[31] D. Canright and Lejla Batina. 2008. A very compact "perfectly masked" S-box for AES. In *Proceedings of the 6th International Conference on Applied Cryptography and Network Security*. 446–459.

[32] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain. 2012. Higher-order masking schemes for S-boxes. In *Proceedings of the 19th International Workshop Fast Software Encryption*. 366–384.

[33] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. 2002. Template attacks. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*. 13–28.

[34] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2019. Quantifying the information leakage in cache attacks via symbolic execution. *ACM Transactions on Embedded Computing Systems* 18, 1 (2019), 7:1–7:27.

[35] Sudipta Chattopadhyay and Abhik Roychoudhury. 2018. Symbolic verification of cache side-channel freedom. *IEEE Transactions on CAD of Integrated Circuits and Systems* 37, 11 (2018), 2812–2823.

[36] Ricardo Chaves, Lukasz Chmielewski, Francesco Regazzoni, and Lejla Batina. 2018. SCA-resistance for AES: How cheap can we go? In *Proceedings of the 10th International Conference on Cryptology in Africa.* 107–123.
[37] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise detection of side-channel vulnerabilities using quantitative Cartesian Hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 875–890.
[38] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Precise cache timing analysis via symbolic execution. In *Proceedings of the IEEE Symposium on Real-Time and Embedded Technology and Applications.* 293–304.
[39] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. 2000. Differential power analysis in the presence of hardware countermeasures. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems.* 252–263.
[40] Scott Contini, Ronald L. Rivest, Matthew J. B. Robshaw, and Yiqun Lisa Yin. 1999. Improved analysis of some simplified variants of RC6. In *Proceedings of the 6th International Workshop on Fast Software Encryption.* 1–15.
[41] Jean-Sébastien Coron. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems.* 292–302.
[42] Jean-Sébastien Coron. 2017. High-order conversion from boolean to arithmetic masking. In *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems.* 93–114.
[43] Jean-Sébastien Coron. 2018. Formal verification of side-channel countermeasures via elementary circuit transformations. In *Proceedings of the 16th International Conference on Applied Cryptography and Network Security.* 65–82.
[44] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. 2012. Conversion of security proofs from one leakage model to another: A new issue. In *Proceedings of the 3rd International Workshop on Constructive Side-Channel Analysis and Secure Design.* 69–81.
[45] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. 2015. Conversion from arithmetic to Boolean masking with logarithmic complexity. In *Proceedings of the 22nd International Workshop on Fast Software Encryption.* 130–149.
[46] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. 2014. Secure conversion between Boolean and arithmetic masking of any order. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems.* 188–205.
[47] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. 2007. Side channel cryptanalysis of a higher order masking scheme. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems.* 28–44.
[48] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. 2013. Higher-order side channel security and mask refreshing. In *Proceedings of the 20th International Workshop on Fast Software Encryption.* 410–424.
[49] Jean DaRolt, Amitabh Das, Santosh Ghosh, Giorgio Di Natale, Marie-Lise Flottes, Bruno Rouzeyre, and Ingrid Verbauwhede. 2012. Scan attacks on side-channel and fault attack resistant public-key implementations. *Journal of Cryptographic Engineering* 2, 4 (2012), 207–219.
[50] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 337–340.
[51] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security* 18, 1 (2015), 4:1–4:32.
[52] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. 2019. Unifying leakage models: From probing attacks to noisy leakage. *Journal of Cryptology* 32, 1 (2019), 151–177.
[53] Hassan Eldib and Chao Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *Proceedings of the 26th International Conference on Computer Aided Verification.* 114–130.
[54] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014), 11.
[55] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. SMT-based verification of software countermeasures against side-channel attacks. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 62–77.
[56] Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of fault-attack countermeasures for cryptographic circuits. In *Proceedings of the 28th International Conference Computer Aided Verification.* 343–363.
[57] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. 2017. Composable masking schemes in the presence of physical defaults and the robust probing model. *IACR Cryptology ePrint Archive* 2017 (2017), 711.
[58] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. 2018. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 3 (2018), 89–120.
[59] Pengfei Gao, Hongyi Xie, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. 2020. Formal verification of masking countermeasures for arithmetic programs. *IEEE Transactions on Software Engineering* (2020).

[60] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. 2019. Quantitative verification of masked arithmetic programs against side-channel attacks. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 155–173.

[61] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Transactions on Software Engineering and Methodology* 28, 3 (2019), 16:1–16:32.

[62] Daniel Genkin, Adi Shamir, and Eran Tromer. 2017. Acoustic cryptanalysis. *Journal of Cryptology* 30, 2 (2017), 392–443.

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

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

[65] Louis Goubin and Jacques Patarin. 1999. DES and differential power analysis (the "duplication" method). In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*. 158–172.

[66] Philipp Grabher, Johann Großschädl, and Dan Page. 2007. Cryptographic side-channels from low-power cache memory. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*. 170–184.

[67] Hannes Groß, Stefan Mangard, and Thomas Korak. 2016. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. IACR Cryptology ePrint Archive. Report 2016/486 (2016).

[68] Hannes Groß and Stefan Mangard. 2018. A unified masking approach. *Journal of Cryptographic Engineering* 8, 2 (2018), 109–124.

[69] Hannes Groß, Stefan Mangard, and Thomas Korak. 2017. An efficient side-channel protected AES implementation with arbitrary protection order. In *Proceedings of the RSA Conference Cryptographers' Track*. 95–112.

[70] Hannes Groß, David Schaffenrath, and Stefan Mangard. 2017. Higher-order side-channel protected implementations of KECCAK. In *Proceedings of the Euromicro Conference on Digital System Design*. 205–212.

[71] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 377–388.

[72] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. 2011. Machine learning in side-channel analysis: A first study. *Journal of Cryptographic Engineering* 1, 4 (2011), 293–302.

[73] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. 2019. Fully automated differential fault analysis on software implementations of block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 3 (2019), 1–29.

[74] Michael Hutter and Michael Tunstall. 2019. Constant-time higher-order boolean-to-arithmetic masking. *Journal of Cryptographic Engineering* 9, 2 (2019), 173–184.

[75] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private circuits: Securing hardware against probing attacks. In *Proceedings of the 23rd Annual International Cryptology Conference*. 463–481.

[76] Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. 2002. Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA. In *Proceedings of the4th International Workshop on Cryptographic Hardware and Embedded Systems, Revised Papers*. 129–143.

[77] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. 2018. Differential power analysis of XMSS and SPHINCS. In *Proceedings of the 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*. 168–188.

[78] Anja F. Karl, Robert Schilling, Roderick Bloem, and Stefan Mangard. 2019. Small faults grow up - verification of error masking robustness in arithmetically encoded programs. In *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation*. 183–204.

[79] HeeSeok Kim, Seokhie Hong, and Jongin Lim. 2011. A fast and provably secure higher-order masking of AES S-Box. In *Proceedings of the 13th International Workshop on Cryptographic Hardware and Embedded Systems*. 95–107.

[80] Yongdae Kim and Haengseok Ko. 2013. Using principal component analysis for practical biasing of power traces to improve power analysis attacks. In *Proceedings of the 16th International Conference on Information Security and Cryptology*. 109–120.

[81] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference*. 104–113.

[82] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference*. 388–397.

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

[84] Xuejia Lai and James L. Massey. 1990. A proposal for a new block encryption standard. In *Proceedings of the Workshop on the Theory and Application of of Cryptographic Techniques*. 389–404.

[85] Junrong Liu, Yu Yu, François-Xavier Standaert, Zheng Guo, Dawu Gu, Wei Sun, Yijie Ge, and Xinjun Xie. 2015. Small tweaks do not help: Differential power analysis of MILENAGE implementations in 3G/4G USIM cards. In *Proceedings of the 20th European Symposium on Research in Computer Security*. 468–480.

[86] Chao Luo, Yunsi Fei, and David R. Kaeli. 2018. Effective simple-power analysis attacks of elliptic curve cryptography on embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*. 115.

[87] Rauf Mahmudlu, Valentina Banciu, Lejla Batina, and Ileana Buhan. 2016. LDA-based clustering as a side-channel distinguisher. In *Proceedings of the 12th International Workshop on Radio Frequency Identification and IoT Security*. 62–75.

[88] Stefan Mangard. 2004. Hardware countermeasures against DPA? A statistical analysis of their effectiveness. In *Proceedings of the RSA Conference Cryptographers' Track*. 222–235.

[89] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer.

[90] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. 2005. Successfully attacking masked AES hardware implementations. In *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems*. 157–171.

[91] Dimitrios Mavroeidis, Lejla Batina, Twan van Laarhoven, and Elena Marchiori. 2012. PCA, eigenvector localization and clustering for side-channel attacks on cryptographic hardware devices. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*. 253–268.

[92] Thomas S. Messerges. 2000. Securing the AES finalists against power analysis attacks. In *Proceedings of the 7th International Workshop on Fast Software Encryption*. 150–164.

[93] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. 2011. Pushing the limits: A very compact and a threshold implementation of AES. In *Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 69–88.

[94] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems*. 58–75.

[95] Yongchuan Niu, Jiawei Zhang, An Wang, and Caisen Chen. 2019. An efficient collision power attack on AES encryption in edge computing. *IEEE Access* 7 (2019), 18734–18748.

[96] Yossef Oren, Ofir Weisse, and Avishai Wool. 2014. A new framework for constraint-based probabilistic template side channel attacks. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems*. 17–34.

[97] Yossef Oren and Avishai Wool. 2016. Side-channel cryptographic attacks using pseudo-boolean optimization. *Constraints* 21, 4 (2016), 616–645.

[98] Siddika Berna Örs, Elisabeth Oswald, and Bart Preneel. 2003. Power-analysis attacks on an FPGA - first experimental results. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems*. 35–50.

[99] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. 2006. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *Proceedings of the RSA Conference Cryptographers' Track*. 192–207.

[100] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. 2005. A side-channel analysis resistant description of the AES S-box. In *Proceedings of the 12th International Workshop on Fast Software Encryption*. 413–423.

[101] Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz. 2017. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *Proceedings of the 6th International Workshop on Security Proofs for Embedded Systems*.

[102] Inès Ben El Ouahma, Quentin L. Meunier, Karine Heydemann, and Emmanuelle Encrenaz. 2019. Side-channel robustness analysis of masked assembly codes using a symbolic approach. *Journal of Cryptographic Engineering* 9, 3 (2019), 231–242.

[103] Elif Ozgen, Louiza Papachristodoulou, and Lejla Batina. 2016. Template attacks using classification algorithms. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust*. 242–247.

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

[105] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of adaptive side-channel attacks. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium*. 328–342.

[106] Stjepan Picek, Kostas Papagiannopoulos, Baris Ege, Lejla Batina, and Domagoj Jakobovic. 2014. Confused by confusion: Systematic evaluation of DPA resistance of various S-boxes. In *Proceedings of the 15th International Conference on Cryptology in India*. 374–390.

[107] Emmanuel Prouff and Matthieu Rivain. 2013. Masking against side-channel attacks: A formal security proof. In *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 142–159.

[108] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. 2009. Statistical analysis of second order differential power analysis. *IEEE Transactions on Computers* 58, 6 (2009), 799–811.

[109] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. 2019. Generic side-channel attacks on CCA-secure lattice-based PKE and KEM schemes. *IACR Cryptology ePrint Archive* 2019 (2019), 948.

[110] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating masking schemes. In *Proceedings of the 35th Annual Cryptology Conference.* 764–783.

[111] Matthieu Rivain and Emmanuel Prouff. 2010. Provably secure higher-order masking of AES. In *Proceedings of the 12th International Workshop on Cryptographic Hardware and Embedded Systems.* 413–427.

[112] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. 2020. A power side-channel attack on the CCA2-secure HQC KEM. *IACR Cryptology ePrint Archive* 2020 (2020), 910.

[113] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. 2019. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Proceedings of the 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography.* 534–564.

[114] Kai Schramm and Christof Paar. 2006. Higher order masking of the AES. In *Proceedings of the RSA Conference Cryptographers' Track.* 208–225.

[115] François-Xavier Standaert. 2017. How (not) to use Welch's t-test in side-channel security evaluations. *IACR Cryptology Eprint Archive* 2017 (2017), 138.

[116] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: A cache timing analysis framework via LLVM transformation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 904–907.

[117] Jiehui Tang, Yongbin Zhou, Hailong Zhang, and Shuang Qiu. 2015. Higher-order masking schemes for simon. In *Proceedings of the 17th International Conference on Information and Communications Security.* 379–392.

[118] David A. Wagner. 2004. Cryptanalysis of a provably secure CRT-RSA algorithm. In *Proceedings of the 11th ACM Conference on Computer and Communications Security.* 92–97.

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

[120] Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating power side channels during compilation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 590–601.

[121] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying cache-based timing channels in production software. In *Proceedings of the 26th USENIX Security Symposium.* 235–252.

[122] Weijia Wang, Yu Yu, and François-Xavier Standaert. 2019. Provable order amplification for code-based masking: How to avoid non-linear leakages due to masked operations. *IEEE Transactions on Information Forensics and Security* 14, 11 (2019), 3069–3082.

[123] Weijia Wang, Yu Yu, François-Xavier Standaert, Junrong Liu, Zheng Guo, and Dawu Gu. 2018. Ridge-based DPA: Improvement of differential power analysis for nanoscale chips. *IEEE Transactions on Information Forensics and Security* 13, 5 (2018), 1301–1316.

[124] Leo Weissbart, Stjepan Picek, and Lejla Batina. 2019. One trace is all it takes: Machine learning-based side-channel attack on EdDSA. In *Proceedings of the 9th International Conference on Security, Privacy, and Applied Cryptography Engineering.* 86–105.

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

[126] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 802–815.

[127] Yinhao Xiao, Yizhen Jia, Chun-Chi Liu, Xiuzhen Cheng, Jiguo Yu, and Weifeng Lv. 2019. Edge computing security: State of the art and challenges. *Proceedings of the IEEE* 107, 8 (2019), 1608–1631.

[128] Jiaming Xu, Ao Fan, Minyi Lu, and Weiwei Shan. 2018. Differential power analysis of 8-bit datapath AES for IoT applications. In *Proceedings of the17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering.* 1470–1473.

[129] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In *Proceedings of the 30th International Conference on Computer Aided Verification.* 157–177.

Received March 2020; revised September 2020; accepted October 2020