Compositional Verification of First-Order Masking Countermeasures against Power Side-Channel Attacks

PENGFEI GAO, Bytedance, China
FU SONG*, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
TAOLUE CHEN, Birkbeck, University of London, UK

Power side-channel attacks allow an adversary to efficiently and effectively steal secret information (e.g., keys) by exploiting the correlation between secret data and run-time power consumption, hence posing a serious threat to software security, in particular, cryptographic implementations. Masking is a commonly used countermeasure against such attacks, which breaks the statistical dependence between secret data and side-channel leaks via randomization. In a nutshell, a variable is represented by a vector of shares armed with random variables, called masking encoding, on which cryptographic computations are performed. While compositional verification for the security of masked cryptographic implementations has received much attention because of its high efficiency, existing compositional approaches either use implicitly fixed pre-conditions which may not be fulfilled by state-of-the-art efficient implementations, or require user-provided hard-coded pre-conditions which is time-consuming and highly non-trivial, even for expert. In this paper, we tackle the compositional verification problem of first-order masking countermeasures, where first-order means that the adversary is allowed to access only one intermediate computation result. Following the literature, we consider countermeasures given as gadgets, that are special procedures whose inputs are masking encodings of variables. We introduce a new security notion parameterized by an explicit pre-condition for each gadget, and composition rules for reasoning about masking countermeasures against power side-channel attacks. We propose accompanying efficient algorithms to automatically infer proper pre-conditions, based on which our new compositional approach can efficiently and automatically prove security for masked implementations. We implement our approaches as a tool MaskCV and conduct experiments on publicly available masked cryptographic implementations including 10 different full AES implementations. The experimental results confirm the effectiveness and efficiency of our approach.

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

Additional Key Words and Phrases: Formal verification, Compositional verification, Cryptographic programs, Side-channel attacks, Masking Countermeasures

ACM Reference Format:
Pengfei Gao, Fu Song, and Taolue Chen. 2023. Compositional Verification of First-Order Masking Countermeasures against Power Side-Channel Attacks. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 38 pages. https://doi.org/10.1145/3635707

*Corresponding author

Authors' addresses: Pengfei Gao, Bytedance, Beijing, China, gaopengfei.se@bytedance.com; Fu Song, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, songfu@ios.ac.cn; Taolue Chen, Birkbeck, University of London, London, UK, WC1E 7HX, t.chen@bbk.ac.uk.

1049-331X/2023/1-ART1
https://doi.org/10.1145/3635707

1 INTRODUCTION

Cryptographic programs have become an indispensable part of contemporary software systems. Practical implementations of cryptographic algorithms unfortunately suffer from side-channel attacks which exploit the statistical dependence between the secret (e.g., keys) and side-channel information (e.g., power consumption [72] and execution time [71]) to effectively recover the secret. In this paper, we focus on power side-channel attacks which have been shown to be successful in practice. For instance, DES [35, 72], AES [84, 94, 97], RSA [56], elliptic curve cryptography [36, 64, 74, 80], and post-quantum cryptography [67, 86, 89] have been the victim. Essentially, power side-channel attacks exploit the correlation between the run-time power consumption of a device executing cryptographic implementations and the secret. The adversary can effectively deduce the results of chosen intermediate computations by statistically analyzing the power consumption traces based on the fact that the power consumption of CMOS transistors (e.g., register) for storing signals 1 and 0 is typically different. When the deduced intermediate computation results rely upon the secret, the adversary can further infer the value of the secret from the intermediate computations and their results. For example, consider the statement $x = k \oplus p$, where k is the secret, p is some public input, and $\oplus$ is the exclusive-OR operation. Obviously, the value of x depends on the value of k for any fixed value of p. For instance, fixing $p = 0$, the probability of $x = 1$ is 100% if $k = 1$ and the probability of $x = 0$ is 100% if $k = 0$. Moreover, the power consumption of a device executing this statement is correlated with the value of x (e.g., the power consumption of the CMOS transistor such as register for storing x varies with the value of x). As a result, the adversary can choose p, and statistically analyze the power consumption traces to deduce the value of x, which leads to the disclosure of the secret k by $x \oplus p$. To effectively and efficiently deduce the secret in real-world cryptographic implementations via power side-channel information, the adversary often choose invertible intermediate computations, e.g., input or output of Sbox in AES, so that the secret can be quickly recovered from these intermediate computations and their results.

Masking is an effective countermeasure against power side-channel attacks, aiming at breaking the statistical dependence between the secret and power consumption via randomization [63]. Typically, an order-d masking scheme splits the secret into d+1 shares such that the joint distribution of any d shares is (statistically) independent of the secret, thus, the adversary cannot infer any information of the secret by observing the values of any d shares via power side-channels. For example, the secret k can be split into two shares $k \oplus r$ and r via first-order masking, where r is a uniformly sampled value, and $(k \oplus r, r)$ forms a masking encoding of the secret k. Since the distributions of both $k \oplus r$ and r are independent of the secret k, the adversary cannot infer the information of the secret k by observing the value of $k \oplus r$. In contrast, the distribution of $k \wedge r$ depends upon the secret k, namely, the probability of $k \wedge r$ being 1 is 50% if $k = 1$ but is 0% if $k = 0$. The adversary can infer the value of k by observing the value of $k \wedge r$. The main challenge is to develop an efficient and secure implementation of each cryptographic algorithm f using masking which performs computations on the secret shares of the original input x, and produces output shares from which the desired output $f(x)$ can be recovered. As efficiency is a major concern in, e.g., resource-limited devices [20], various new masked implementations for finite-field multiplication, a key building block to implement most cryptographic algorithms, have been proposed recently [8, 11, 14, 15, 26, 58, 68, 93]. These more recent implementations require less randomness and/or operations than the original one proposed by [63].

Clearly, a masked implementation must meet some security requirement, e.g., the *order-d probing security* [63] which asserts that the joint distribution of any d observable variables to the adversary should be independent of the secret data. For instance, the masked implementation of the statement $x = k \oplus p$ via order-d masking is $r_{d+1} = k \oplus r_1 \oplus \cdots \oplus r_d;\ x = r_{d+1} \oplus p$, where $r_1, \cdots, r_d$ are uniformly

sampled values, $(r_1, \cdots, r_d, r_{d+1})$ forms a masking encoding of the secret k, the desired result $k \oplus p$ can be obtained from the output encoding $(r_1, \cdots, r_d, x)$ via computing $r_1 \oplus \cdots \oplus r_d \oplus x$. This masked implementation is order-d probing secure because the joint distribution of any d observable variables from $\{r_1, \cdots, r_d, r_{d+1}, x\}$ is uniform, thus independent of the secret k. However, it is *not* order-$(d+1)$ probing secure, because the joint distribution of some $(d+1)$ observable variables (e.g., $\{r_1, \cdots, r_d, r_{d+1}\}$ or $\{r_1, \cdots, r_d, x\}$) depends on the secret k. Indeed, $r_1 \oplus \cdots \oplus r_d \oplus r_{d+1}$ is the same as the secret k and $r_1 \oplus \cdots \oplus r_d \oplus x$ is the same as $k \oplus p$ while p is known to the adversary. Designing masked implementations is however labor-intensive and error-prone. Indeed, some published masked implementations [87, 90] were later shown to be vulnerable against power side-channel attacks under the same leakage model and masking order [40, 41]. To address this concern, various formal verification approaches have been proposed, which form an important subarea of software security research. These approaches can be largely be divided into two categories, i.e., non-compositional (i.e., intra-procedural) and compositional (i.e., inter-procedural) approaches [9, 18, 53]. Following the naming convention in the literature on masked implementations of cryptographic algorithms, procedures are called *gadgets* in this work, because the input parameters and return value of the procedures are often masking encodings. Furthermore, a gadget is referred to as *simple gadget* if it does not contain any call statements, otherwise it is referred to as *composite gadget*.

Non-compositional approaches take a simple gadget as input, and verify the observable variables one by one or set by set. The underlying techniques include symbolic analysis [21, 38, 76, 77, 81, 91, 92], SAT/SMT-based analysis [23, 47–50], BDD analysis [69], and hybrid approaches [52, 54, 55, 98]. Note that [21, 23, 47–50, 55, 69, 98] focus exclusively on the bitwise logical operations (e.g., and, or, exclusive-or) and cannot directly handle *arithmetic* operations (e.g., addition, subtraction).

To verify a composite gadget using non-compositional approaches, all the gadget calls must be inlined first. However, inlining gadget calls will produce plenty of observable variables with large computation expressions, leading to the inefficient verification. Compositional approaches are thus proposed by leveraging *stronger* security notions [9, 10, 12, 16–19, 25, 26, 30, 31, 69] to directly analyze composite gadgets without inlining. All these approaches implicitly specify a fixed pre-condition for each gadget which intuitively restricts the number or positions of input shares. Unfortunately, many modern efficient implementations (e.g., [14, 20]) do not satisfy these implicitly imposed pre-conditions, creating a gap between formally provable probing security and practically used (efficient) masked implementations. To address this issue, an assume-guarantee based compositional approach has been proposed [53] for verifying first-order security of arithmetic masked programs. However, hard-coded (non-standard) pre-conditions, specifying the relation between support variables of the computations of formal arguments must be provided by users, which impedes fully automated compositional verification. While inferring specifications and contracts of procedures from source code is a well researched area in formal safety verification, e.g., [3, 5, 62], they are not applicable to compositional verification of security for masked implementations.

In this work, we fill this gap for first-order probing security, which is the focus of a large body of research work particularly in software security [47–50, 53, 54, 76, 77, 81, 91, 92, 98]. Following the literature, we consider masked implementations that are given as gadgets and composition of gadgets, that are special procedures whose inputs are masking encodings of variables. Specifically, we propose a new security notion parameterized by an explicit pre-condition $\mathbb{I}$ for each gadget, called $\mathbb{I}$-*Non-Interference*, based on which we present a compositional verification approach that is applicable to those efficient masked implementations without user-provided pre-conditions. The pre-condition $\mathbb{I}$ of a gadget g is a set of variable sets, and is used to characterize the first-order probing security of the gadget g, namely, each observable variable of the gadget g can be perfectly simulated by a variable set from $\mathbb{I}$. Note that the pre-condition $\mathbb{I}$ can vary with gadgets. For some gadgets, the formal parameters (i.e., input shares) are sufficient to simulate the internal

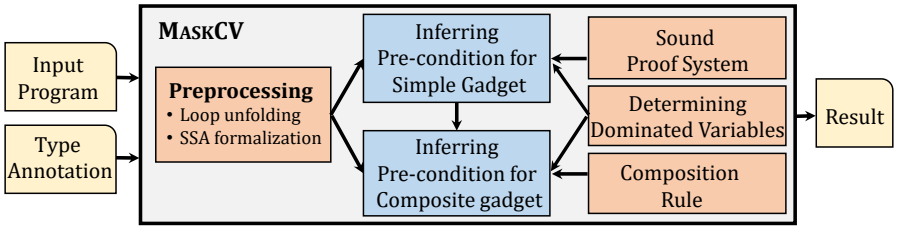

Fig. 1. Overview of our method

variables, otherwise, internal variables would be added into $\mathbb{I}$. (The worst case would be that all the internal variables are added into $\mathbb{I}$, which amounts to inlining the gadget call.) The flexibility of $\mathbb{I}$ gives wider applicability to generic and efficient masked implementations while fixed pre-conditions [9, 10, 12, 16–19, 25, 26, 30, 31, 69] fail.

Fig. 1 shows an overview of our method, consisting of two main steps, namely, preprocessing and verification. Given a masked program and its security type annotation of input parameters, the preprocessing step automatically unfolds loops in the program and transforms the program into its intermediate representation in the static-single assignment (SSA) form. The verification step infers pre-conditions of gadgets based on which first-order probing security is verified. We propose efficient algorithms for inferring pre-conditions of both simple and composite gadgets. The pre-conditions of simple gadgets are saturated during judgement inference via a sound proof system until they become sufficient. The pre-conditions of composite gadgets are inferred by leveraging a novel composition rule which composes pre-conditions of the called gadgets. More specifically, the verification process starts by inferring the pre-condition $\mathbb{I}$ for the `main` gadget. The gadget call statements in the `main` gadget are iteratively traversed during which the pre-conditions of the called gadgets are computed by recursively calling the algorithms for inferring pre-conditions of simple and/or composite gadgets. To facilitate the processing of the subsequent gadget call statements, the resulting pre-conditions of gadgets are cached. Finally, the pre-conditions of called gadgets are composed according to the composition rule, resulting in the pre-condition $\mathbb{I}$ for the `main` gadget. Finally, to check whether a program is first-order probing secure or not, i.e., each observable variable is statistically independent of the secret inputs, we can check the pre-condition $\mathbb{I}$ of the `main` gadget, where the pre-condition $\mathbb{I}$ is expected to be much smaller than the size of observable variables after inlining. Moreover, we observe that a variable (called *dominated variable*) whose computation is perfectly masked by a random variable (called *dominant variable*) could also be used to mask other variables. For instance, the variable x with $x = k \oplus r$ is perfectly masked by the random variable r as the distribution of x is uniform for any fixed value of k, thus x is a dominated variable and r is its dominant variable. This observation often allows us to quickly conclude that any dominated variable is statistically independent of the secret inputs, thus is leveraged in the algorithms for inferring pre-conditions of simple and composite gadgets to reduce the size of pre-conditions and improve verification efficiency.

We implement our approach as a tool MaskCV, and conduct extensive experiments on several publicly available masked implementations [53] including 10 different masked implementations of full AES. The experimental results confirm the efficacy of our approach, e.g., MaskCV can prove each of the 10 masked implementations of full AES in no more than 0.04 seconds. Compared with the compositional approach QMVerif [53], our approach performs significantly better when no pre-conditions are provided to QMVerif, and achieves competitive efficiency when all the user-defined pre-conditions are provided to QMVerif. Indeed, QMVerif takes 197–3505 seconds

to prove each of 10 masked implementations of full AES when no pre-conditions are provided. (To the best of our knowledge, QMVerif is the only compositional tool that can verify these benchmarks.) Compared with the state-of-the-art non-compositional approaches `SILVER` [69] and `LeakageVerif` [77], our approach is significantly more efficient and effective. For instance, `SILVER` does not support masked implementations of full AES which involve arithmetic operations and runs out of time or memory (6 hours, 256GB RAM) on two Boolean implementations of AES Sbox which can be proved by MaskCV in 0.01 seconds; `LeakageVerif` runs out of time or memory on 6 out of 10 masked implementations of full AES and takes at least 559 seconds for any of the other 4 masked implementations.

The main contributions are summarized as follows.

- We introduce a new security notion, called $\mathbb{I}$-Non-Interference, which is parameterized by an explicit and variable pre-condition $\mathbb{I}$. We also present a composition rule for compositional reasoning about first-order probing security of efficient masked implementations that cannot be handled by existing fully automated compositional verification approaches [9, 10, 12, 16–19, 25, 26, 30, 31, 69].
- We propose efficient algorithms for inferring pre-conditions of both simple and composite gadgets, leading to an efficient and fully automated composition verification approach for first-order probing security of arithmetic masked programs. Thus, providing hard-coded (non-standard) pre-conditions in the assume-guarantee based compositional approach [53], which is time-consuming and highly non-trivial even for expert, can be avoided.
- We propose an efficient approach to determine dominated variables that can reduce the size of pre-conditions and thus significantly improves the verification efficiency. It paves the way for integrating the power side-channel security verification into the software development process and continual verification during development.
- We implement the proposed techniques in a tool MaskCV and conduct extensive experiments on publicly available cryptographic benchmarks, which confirm the effectiveness and efficiency of our approach. In particular, on the 10 masked implementations of full AES, the automatically determined dominated variables improve the verification efficiency by 4 orders of magnitude; and MaskCV shows almost 5–6 orders of magnitude improvement with respect to `LeakageVerif` and QMVerif without user-defined pre-conditions.

Our work is important for the software engineering community. First, it can be readily used by cryptographers and software developers of cryptographic algorithms to ensure the (first-order) probing security of their implementations against power side-channel attacks. Cryptographic algorithms have been extensively used in various applications, ranging from Blockchain, Internet of Things, edge computing, to smart devices. It is important to ensure that software systems are robust against power side-channel attacks, particularly for security-critical applications, where our work would play a crucial role. Moreover, MaskCV is highly automated and can be fully integrated into the software development process, so potentially would serve a larger group of software developers. Furthermore, the verification techniques we use, such as proof system and compositional reasoning, represent the state-of-the-art research in quality assurance, an important area of software engineering.

Outline. Section 2 gives the preliminary of this work (i.e., cryptographic programs considered in this work, threat model, leakage model and security notions) and a running example for demonstration. We propose the new security notion $\mathbb{I}$-Non-Interference and its composition rule in Section 3. In Section 4, we introduce the concept of dominated variables, an approach to determine dominated variables and the application of dominated variables for inferring pre-conditions. Section 5 presents the algorithms for inferring pre-conditions for both simple and composite gadgets. Section 6 reports

Table 1. Notations.

Notation	Description	Notation	Description
$\mathbb{I}$	set of variable sets used as pre-condition	$\mathbb{F}$	underlying domain $\{0, \cdots, 2^n - 1\}$
x	scalar variable	$\vec{x}$	vector of variables used as masking encoding of x
$\vec{x}[i]$	the i-th share of the encoding $\vec{x}$	X_k	set of private variables of a program
$G_c/G_s/G_{en}$	simple/composite/encoding gadget	X_p	set of public variables of a program
g_{in}/P_{in}	inlined version of the gadget g/program P	$\mathcal{E}(x)$	computation of a variable x
$\mathrm{Sub}(x)$	set of the sub-expressions of $\mathcal{E}(x)$	$\mathrm{Var}(\mathcal{E}(x))$	set of support variables of $\mathcal{E}(x)$
X^g_{en}	set of input encodings of the gadget g	$X^g_{en}[i]$	the i-th input encoding of the gadget g
T_g	summary of the gadget g	η	valuation for a set of variables
X^g_r	set of random variables defined in the gadget g	$\mathcal{D}(V)$	set of the joint distributions of variables in V
X^g_o	set of output variables defined in the gadget g	$\mathrm{DomR}(x)$	set of random dominant variables of the variable x
X^g_a	set of formal parameters of the gadget g	$[\![\mathcal{E}(x)]\!]^P_\eta$	probability distribution of x in program P under η
X^g	set of internal variables defined in the gadget g	$[\![X]\!]^P_\eta$	joint distribution of variables of X in P under η

our experimental results on various publicly available masked cryptographic implementations. We discuss related work in Section 7 and conclude the paper in Section 8. The source code of our tool, raw data of experimental results and benchmarks are available at [1].

2 PRELIMINARY

In this section, we first introduce the syntax and semantics of the program considered in this work, then recap the threat model, leakage model and related security notions, and finally present a running example for demonstration of our approach.

Notations. For convenient reference, we summarize the notations in Table 1.

2.1 Cryptographic Programs

Given a positive integer n, let $\mathbb{F}$ be the domain $\{0, \cdots, 2^n - 1\}$ (e.g. the finite-field $\mathbb{GF}(2^n)$). We use the syntax shown in Fig. 2 to describe masked implementations. Symbols such as x, a, a_1, a_m are scalar variables, symbols with arrowed overline such as $\vec{b}, \vec{a}_1, \vec{a}_m$ represent vectors of variables. A vector $\vec{x}$ of variables is used to represent the shares of a scalar variable x, thus called a (masking) *encoding* of the scalar variable x in this work. We denote by $\vec{x}[i]$ the i-th variable in the encoding $\vec{x}$, that is one share of the scalar variable x and by $\bigoplus \vec{x}$ the XOR of all the shares of the encoding $\vec{x}$. A masked implementation always has a security parameter d denoting order-d masking, and the size of each encoding is typically $d + 1$. We explain the syntax rules in Fig. 2 from bottom to top.

A program is defined as a sequence of gadget definitions with a `main` gadget, where scalar variables $a_1, \cdots, a_m$ are its formal parameters. We assume that the formal parameters of the `main` gadget are classified into their types, either private or public, provided by the users. We denote by X_k the set of private variables and X_p the set of public variables, so that $X_k \uplus X_p = \{a_1, \cdots, a_m\}$. The `main` gadget has a sequence of (masking) encoding call statements `enstmt` and gadget call statements `gstmt`, ending with a return statement `return` $\vec{b}$ which gives the output encoding $\vec{b}$.

An encoding call statement (i.e., `enstmt`) generates an encoding $\vec{x}$ of a scalar variable x by calling an encoding gadget (i.e., $\vec{x} = f(x)$). An encoding gadget (i.e. $f(a)\{$`stmts; return` $\vec{a}$; $\}$) is defined as a common procedure, which takes a scalar variable a as input and outputs the encoding of a, i.e., $\vec{a}$. An encoding gadget computes the desired encoding via a sequence of statements that can express different encoding schemes. Normally, the first d variables in an encoding (i.e., $\vec{a}[1] \cdots \vec{a}[d]$) are generated by random sampling, while $\vec{a}[d + 1]$ is a computation of $\vec{a}[1], \cdots, \vec{a}[d]$ and the input

Constant:	$\mathbb{F} \ni c$	::=	n-bit constant
Operation:	$\mathbf{Op} \ni \circ$	::=	$\wedge \mid \vee \mid \oplus \mid - \mid + \mid \times \mid \odot$
Expression:	e	::=	$c \mid x \mid \neg e \mid e \ll c \mid e \gg c \mid e \circ e$
Statements:	`stmts`	::=	$x = e; \mid r = \$; \mid$ `stmts`$^+$
Simple gadget:	G_s	::=	$g(\vec{a}_1, \cdots, \vec{a}_m)$
			{`stmts return` $\vec{b}$;}
Composite gadget:	G_c	::=	$g(\vec{a}_1, \cdots, \vec{a}_m)$
			{`gstmt`$^+$ `return` $\vec{b}$;}
Gadget call:	`gstmt`	::=	$\vec{x} = g(\vec{y}_1, \cdots, \vec{y}_m);$
Encoding gadget:	G_{en}	::=	$f(a)$ {`stmts return` $\vec{a}$;}
Encoding call:	`enstmt`	::=	$\vec{x} = f(x);$
Program:	P	::=	$G_{en}^+ \; G_s^+ \; G_c^*$
			`main`$(a_1, \cdots, a_m)$
			{`enstmt`$^+$ `gstmt`$^+$ `return` $\vec{b}$;}

Fig. 2. Syntax of Programs.

variable a. *Boolean masking* is always used in implementing masked cryptographic algorithms which only use bitwise logical operations (see below) and $\vec{a}[d+1] = \vec{a}[1] \oplus \vec{a}[2] \oplus \cdots \oplus \vec{a}[d] \oplus a$. Masked implementations which only use arithmetic operations are called *arithmetic masking*, and they compute $\vec{a}[d+1]$ using modular addition, namely, $\vec{a}[d+1] = \vec{a}[1] + \vec{a}[2] + \cdots + \vec{a}[d] + a$. There are some implementations that use both bitwise logical operations and arithmetic operations for which conversions between Boolean and arithmetic masking [37, 39] are used.

A gadget call statement (i.e., `gstmt`) assigns the result of a gadget call to an internal encoding. The syntax supports two types of gadgets: *simple* (i.e., G_s) and *composite* (i.e., G_c) gadgets, both of which take a list of input encodings $\vec{a}_1, \cdots, \vec{a}_m$ as inputs and return the output encoding $\vec{b}$. A simple gadget only contains a sequence of statements (i.e., `stmts`) without involving any gadget call statements, while a composite gadget only contains a sequence of gadget call statements (i.e., `gstmt`$^+$) with unique labels ℓ (denoting call-sites), both of which ends with a return statement `return` $\vec{b}$. We assume that all the gadgets are given in static single assignment (SSA) form. We denote by X_{en}^g the vector (e.g., $[\vec{a}_1, \cdots, \vec{a}_m]$) of all the input encodings of the gadget g, and by $X_{en}^g[i]$ the i^{th} input encoding (e.g., $\vec{a}_i$). Let X_a^g be the union of all the variables in input encodings of the gadget g, i.e., $X_a^g = \bigcup_{1 \le i \le m} \vec{a}_i$. If g is a `main` gadget or an encoding gadget, X_a^g is the set of input parameters, that are scalar variables. An encoding gadget may be called simple gadget as well, because it does not contain any call statements.

There are two types of statements. One is the common assignment statement of the form $x = e$ which assigns the value of the expression e to the scale variable x. The other is of the form $r = \$$ which assigns a uniformly sampled random value to the scale variable r, thus, r is a *random variable*. Random variables are used to mask private-related variables. We denote by X_r^g and X^g respectively the set of random variables and the set of internal variables (excluding input parameters X_a^g) defined in the gadget g. Similarly, we denote by X_o^g the set of output variables of the gadget g.

An expression e is built up from scale variables and constants with the following operations:

- bitwise logical operations: *and* ($\wedge$), *or* ($\vee$), *exclusive-or* ($\oplus$), *negation* ($\neg$), *left shift* ($\ll$), *right shift* ($\gg$);
- modulo 2^n arithmetic operations: *subtraction* ($-$), *addition* ($+$), *multiplication* ($\times$) for which $\mathbb{F}$ is regarded as the ring $\mathbb{Z}_{2^n}$ of integers modulo 2^n.
- finite-field operation: *multiplication* ($\odot$), for which $\mathbb{F}$ is considered to be the Galois field $\mathbb{GF}(2^n)$.

```
1  Encoding(k){
2    a⃗[1] = $;
3    a⃗[2] = a⃗[1] ⊕ k;
4    return a⃗;  }  // ⨁ a⃗ = k
5  XOR(a⃗, b⃗){
6    for(i = 1; i ≤ 2; i++){
7      c⃗[i] = a⃗[i] ⊕ b⃗[i];}
8    return c⃗;  }  // ⨁ c⃗ = ⨁ a⃗ ⊕ ⨁ b⃗
9  main(k1, k2){
10   a⃗ = Encoding(k1);
11   b⃗ = Encoding(k2);
12   c⃗ = XOR(a⃗, b⃗);
13   return c⃗;  }  // ⨁ c⃗ = k1 ⊕ k2
```

Fig. 3. Encoding is an encoding gadget, XOR is a simple gadget, and main is the main gadget

For a simple gadget g, the *computation* $\mathcal{E}(x)$ of each variable $x \in X_a^g \cup X^g$ is defined as follows:

- If $x \in X_a^g \cup X_r^g$, $\mathcal{E}(x) = x$;
- If $x = e$, $\mathcal{E}(x)$ is obtained by recursively replacing all the occurrences of each variable y in e by its computation $\mathcal{E}(y)$ until no more update can be made.

Intuitively, $\mathcal{E}(x)$ defines the computation of the variable x in terms of input parameters, constants and random variables, thus can be seen as the symbolic value of the variable x. We denote by $\mathsf{Var}(\mathcal{E}(x))$ the set of support variables of the variable x, i.e., the input parameters and random variables that are involved in the computation $\mathsf{Var}(\mathcal{E}(x))$.

For a composite gadget g, to obtain the computation of a variable $x \in X_a^g \cup X^g$, gadget calls should be inlined first. Inlining a gadget call is the same as inlining a procedure call, which replaces the gadget call by the gadget body of the callee, followed by assignments mimicking the return statement, where the local variables are appended with $@\ell$ of the call-site ℓ to avoid name conflict. We denote by $f(\vec{x}_1, \cdots, \vec{x}_m)@\ell$ the corresponding statements after inlining the gadget call $\vec{y} = f(\vec{x}_1, \cdots, \vec{x}_m)$ with call-site ℓ. After recursively inlining all the gadget calls, a composite gadget turns to a simple one. We denote by g_{in} the inlined version of g. Specifically, P_{in} is the inlined version of the program P.

Example 2.1. Fig. 3 presents three gadgets, where Encoding is an encoding gadget, XOR is a simple gadget [63] and main is the main gadget.

The Encoding gadget splits an input k into two shares $\vec{a}[1]$ and $\vec{a}[2]$ via first-order Boolean masking, where $\vec{a}[1]$ is randomly sampled and $\vec{a}[2] = \vec{a}[1] \oplus k$ masks k by XORing $\vec{a}[1]$, resulting in the encoding $\vec{a}$ of the scalar variable k. For the Encoding gadget, we have:

- $X_a^{\mathsf{Encoding}} = \{k\}$, $X_r^{\mathsf{Encoding}} = \{\vec{a}[1]\}$, $X_o^{\mathsf{Encoding}} = X^{\mathsf{Encoding}} = \{\vec{a}[1], \vec{a}[2]\}$,
- $\mathcal{E}(k) = k$ and $\mathcal{E}(\vec{a}[1]) = \vec{a}[1]$ (as $k, \vec{a}[1] \in X_a^{\mathsf{Encoding}} \cup X_r^{\mathsf{Encoding}}$), and $\mathcal{E}(\vec{a}[2]) = \vec{a}[1] \oplus k$.

The XOR gadget performs share-wise XOR, i.e., produces the encoding $(\vec{a}[1] \oplus \vec{b}[1], \vec{a}[2] \oplus \vec{b}[2])$ for the input encodings $\vec{a}$ and $\vec{b}$. We have:

- $X_{en}^{\mathsf{XOR}} = \{\vec{a}, \vec{b}\}$ where $X_{en}^{\mathsf{XOR}}[1] = \vec{a}$ and $X_{en}^{\mathsf{XOR}}[2] = \vec{b}$,
- $X_a^{\mathsf{XOR}} = \{\vec{a}[1], \vec{a}[2], \vec{b}[1], \vec{b}[2]\}$, $X_r^{\mathsf{XOR}} = \emptyset$, $X_o^{\mathsf{XOR}} = X^{\mathsf{XOR}} = \{\vec{c}[1], \vec{c}[2]\}$,
- $\mathcal{E}(x) = x$ for every $x \in X_a^{\mathsf{XOR}}$, and $\mathcal{E}(\vec{c}[i]) = \vec{a}[i] \oplus \vec{b}[i]$ for $i = 1, 2$.

The main gadget consists of two encoding call statements to the Encoding gadget and one gadget call statement to the simple gadget XOR. It takes two secrets k_1 and k_2 as inputs, first computes their encodings via the Encoding gadget and then computes $k_1 \oplus k_2$ via the simple gadget XOR using the encodings $\vec{a}$ and $\vec{b}$ of k_1 and k_2, resulting in the output encoding $\vec{c}$ of $k_1 \oplus k_2$. We have:

- $X_a^{\mathsf{main}} = X_k = \{k_1, k_2\}$, $X_p = \emptyset$,
- $X_r^{\mathsf{main}} = \emptyset$, $X^{\mathsf{main}} = \{\vec{a}[1], \vec{a}[2], \vec{b}[1], \vec{b}[2], \vec{c}[1], \vec{c}[2]\}$, $X_o^{\mathsf{main}} = \{\vec{c}[1], \vec{c}[2]\}$.

```
1  main_in(k_1, k_2){
2    a⃗[1]@10 = $;
3    a⃗[2]@10 = a⃗[1]@10 ⊕ k_1;
4    a⃗[1] = a⃗[1]@10;
5    a⃗[2] = a⃗[2]@10;
6    a⃗[1]@11 = $;
7    a⃗[2]@11 = a⃗[1]@11 ⊕ k_2;
8    b⃗[1] = a⃗[1]@11;
9    b⃗[2] = a⃗[2]@11;
10   c⃗[1]@12 = a⃗[1] ⊕ b⃗[1];
11   c⃗[2]@12 = a⃗[2] ⊕ b⃗[2];
12   c⃗[1] = c⃗[1]@12;
13   c⃗[2] = c⃗[2]@12;
14   return c⃗;  }
```

Fig. 4. The inlined version $\texttt{main}_{in}$ of the main gadget

The inlined version of the main gadget is shown in Fig. 4. We have:

- $X_a^{\texttt{main}_{in}} = X_k = \{k_1, k_2\}$, $X_p = \emptyset$,
- $X_r^{\texttt{main}_{in}} = \{\vec{a}[1]@10, \vec{a}[1]@11\}$, $X_o^{\texttt{main}_{in}} = \{\vec{c}[1], \vec{c}[2]\}$
- $X^{\texttt{main}_{in}} = X^{\texttt{main}} \cup \{\vec{a}[1]@10, \vec{a}[2]@10, \vec{a}[1]@11, \vec{a}[2]@11, \vec{c}[1]@12, \vec{c}[2]@12\}$,
- the computations $\mathcal{E}(x)$ for every $x \in X^{\texttt{main}_{in}}$ can be constructed accordingly, e.g.,

$\mathcal{E}(\vec{c}[1]) = \vec{a}[1]@10 \oplus \vec{a}[1]@11$ and $\mathcal{E}(\vec{c}[2]) = (\vec{a}[1]@10 \oplus k_1) \oplus (\vec{a}[1]@11 \oplus k_2)$. □

Discussion on the program syntax. For convenience, our verification tool supports bounded for-loops (e.g., the for-loop in the gadget XOR) which are automatically and fully unfolded before verification, thus we only present the core language without loops. One may notice that the program syntax has a specific format, where (1) inputs and outputs of both simple and composite gadgets are vectors, (2) encoding and main gadgets take only scalar variables as inputs and return a vector, and (3) both composite and main gadgets only consist of call statements except for the return statement. This specific format is consistent with the design pattern of masked implementations of cryptographic algorithms and is widely adopted in the literature [9–11, 14, 15, 18, 19, 30, 31, 63, 87], thus our approach has a wide application on masked implementations of cryptographic algorithms. In a nutshell, the design process of a masked implementation starts by choosing a masking scheme (e.g., Boolean masking) which is implemented by an encoding gadget, thus an encoding gadget takes a scalar variable as input and produces an encoding of the scalar variable. Then, one will design a masked version for each operation used in the cryptographic algorithm, and the masked version performs the desired computation on the encodings of the operands of the operation. For the sake of code reuse, modularity and maintenance in software development, masked versions of all the operations of cryptographic algorithms are wrapped as gadgets (i.e., functions), thus the inputs and output of a simple or composite gadget are encodings that correspond to the operands and results of the operation, respectively. Finally, each original function in the cryptographic algorithm becomes a composite gadget, where the inputs and output are replaced by the corresponding encodings, and each statement is replaced by a gadget call statement to the corresponding gadget. Similarly, the main function is revised by adding encoding call statements and gadget call statements, resulting in a main gadget.

Semantics. Given a variable set V, we denote by $\eta : V \to \mathbb{F}$ the valuation of V, which maps variables to concrete values and denote by $\mathcal{D}(V)$ the set of all the joint distributions of V.

Let us fix a program P. Given a valuation η of the inputs $X_k \cup X_p$ of the program P, the computation $\mathcal{E}(x)$ of a variable x in the program P_{in} is interpreted as a probability distribution, denoted by $[\![\mathcal{E}(x)]\!]_\eta^P$, where variables in $\mathsf{Var}(\mathcal{E}(x)) \cap (X_k \cup X_p)$ are instantiated by the valuation η and random variables are sampled uniformly. Given a subset of variables X of the program P_{in}, we also denote by $[\![X]\!]_\eta^P$ the joint distribution of variables in X under the valuation η.

Given a gadget g, let $\mu \in \mathcal{D}(X_a^g)$ be a joint distribution of the formal parameters X_a^g. For each $x \in X^g \cup X_a^g$, the computation $\mathcal{E}(x)$ is interpreted as a probability distribution, denoted by $[\![\mathcal{E}(x)]\!]_\mu^g$, where variables in $\mathsf{Var}(\mathcal{E}(x)) \cap X_a^g$ are sampled from μ and $\mathsf{Var}(\mathcal{E}(x)) \cap X_r^g$ are sampled uniformly.

Example 2.2. Consider the variables $\vec{c}[1]$and $\vec{c}[2]$ in the example P shown in Fig. 4. Recall that

$$\mathcal{E}(\vec{c}[1]) = \vec{a}[1]@10 \oplus \vec{a}[1]@11 \text{ and } \mathcal{E}(\vec{c}[2]) = (\vec{a}[1]@10 \oplus k_1) \oplus (\vec{a}[1]@11 \oplus k_2).$$

Then, for any valuation η of the inputs $\{k_1, k_2\}$, the probability distributions $[\![\mathcal{E}(\vec{c}[1])]\!]^P_\eta, [\![\mathcal{E}(\vec{c}[2])]\!]^P_\eta$ and $[\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta$ are defined as:

- $[\![\mathcal{E}(\vec{c}[1])]\!]^P_\eta(0) = [\![\mathcal{E}(\vec{c}[1])]\!]^P_\eta(1) = [\![\mathcal{E}(\vec{c}[2])]\!]^P_\eta(0) = [\![\mathcal{E}(\vec{c}[2])]\!]^P_\eta(1) = \frac{1}{2}$, because $\vec{a}[1]@10$ and $\vec{a}[1]@11$ are random variables;
- if $\eta(k_1) \oplus \eta(k_2) = 0$: $[\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(0,0) = [\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(1,1) = \frac{1}{2}$ and $[\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(0,1) = [\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(1,0) = 0$, because the random variables $\vec{a}[1]@10$ and $\vec{a}[1]@11$ are used in both $\mathcal{E}(\vec{c}[1])$ and $\mathcal{E}(\vec{c}[2])$, causing interference between the values of $\mathcal{E}(\vec{c}[1])$ and $\mathcal{E}(\vec{c}[2])$;
- if $\eta(k_1) \oplus \eta(k_2) = 1$: $[\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(0,1) = [\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(1,0) = \frac{1}{2}$ and $[\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(0,0) = [\![\vec{c}[1], \vec{c}[2]]\!]^P_\eta(1,0)$. □

2.2 Thread Model, Leakage Model and Security Notions

Thread model and leakage model. In this work, we adopt the widely used threat model [47, 48, 53, 76, 77, 81, 98], where the adversary knows the details of the implementation and has the ability to choose the values for the public input variables X_p, but do not know the values of the private input variables X_k, of the program P. Moreover, the adversary may have access to the results of one arbitrary-chosen intermediate computation (i.e., observable variable in P_{in}) via power side-channel information, where all the variables in the program P_{in} except for the private input variables X_k are observable variables to the adversary. Under these assumptions, the adversary's goal is to infer the value of private inputs X_k.

The power side-channel attacks exploit the correlation between power consumption values, rather than the absolute power consumption. In this work, we consider the correlation between power consumption values that comes from the static leakage currents of CMOS transistors, where power consumption volume depends on whether the transistor is on or off, corresponding to the logical 1 and 0 of a bit, respectively. Thus, the value of a variable is proportional to the power consumption of storing the value. For example, if the device executes the statement $a = k \oplus p$, where k is the secret and p is the public, assuming that k and p are Boolean variables and $p = 1$, then $a = 1$ if $k = 0$ and $a = 0$ if $k = 1$. The value of a thus depends on the value of k. The value of a, hence the value of k, can be deduced by analyzing the power consumption of executing the statement $a = k \oplus p$.

Simulatability. To characterize that a set of variables is statistically independent upon another variable set, we introduce the notions of randomized function and simulatability.

A *randomized function* $\pi : \mathbb{F}^{|I|} \to \mathbb{F}^{|O|}$ over two sets of variables I and O is a function such that for any fixed tuple of values $(v_1, \cdots, v_{|I|}) \in \mathbb{F}^{|I|}$, $\pi(v_1, \cdots, v_{|I|})$ is a (joint) distribution of the values of the variables in O.

Given a set of variables I, a set of variables O can be simulated by the set of variables I, called *I-simulatable*, if there exists a randomized function $\pi : \mathbb{F}^{|I|} \to \mathbb{F}^{|O|}$ such that for any fixed tuple of values $(v_1, \cdots, v_{|I|}) \in \mathbb{F}^{|I|}$ and any valuation $\eta : X_a^{\texttt{main}} \to \mathbb{F}$, the (joint) distributions $\pi(v_1, \cdots, v_{|I|})$ and $[\![O]\!]^P_\eta$ are the same when the values of I in the program P are limited to $(v_1, \cdots, v_{|I|})$.

Intuitively, when the values of variables in the set I are fixed, the joint distribution of (the values of) the variables in set O is fixed as well. In other words, knowing the values of the variables in I suffices to simulate the distribution $[\![O]\!]^P_\eta$. For example, consider $y = x \oplus r$ and $y' = x \wedge r$, where x is an input variable and r is a random variable. The variable y is $\emptyset$-simulatable, meaning that the distribution of y can be simulated without knowing the values of any variables. But, y' is

$\{x\}$-simulatable instead of $\emptyset$-simulatable, meaning that the distribution of y can be simulated when knowing the value of x.

Similarly, for a gadget g and a variable set I, a variable set O is *I-simulatable*, if there exists a randomized function $\pi : \mathbb{F}^{|I|} \to \mathbb{F}^{|O|}$ such that for any fixed tuple of values $(v_1, \cdots, v_{|I|}) \in \mathbb{F}^{|I|}$ and any distribution $\mu \in \mathcal{D}(X_a^g)$, the distributions $\pi(v_1, \cdots, v_{|I|})$ and $[\![O]\!]_\mu^g$ are the same when the values of I in the gadget g are limited to $(v_1, \cdots, v_{|I|})$.

It is straightforward to obtain the following proposition.

Proposition 2.3. *Suppose the variable set O is I-simulatable. The following statements hold:*

- *O is I'-simulatable for any I' such that $I \subseteq I'$;*
- *O' is I-simulatable for any O' such that $O' \subseteq O$ or O' is O-simulatable.* □

According to Proposition 2.3, for any variable set O, there always exists a variable set I (e.g., O) such that O is I-simulatable. We will see later that the smaller the size of I the better for proving security. In Section 5.1, we will present a sound proof system for checking simulatability.

Hereafter, for the sake of presentation, a singleton set $\{x\}$ may be directly written as x, e.g., $\{y\}$ is $\{x\}$-simulatable is written as y is x-simulatable.

Probing security. A program P is *first-order probing secure* if each observable variable x in the program P_{in} is X_p-simulatable. Intuitively, the program P is first-order probing secure if the distribution of each observable variable x can be simulated by only knowing the values of public inputs. Thus the distribution of the observable variable x is (statistically) independent of the private inputs X_k and the adversary cannot infer any information of the private inputs X_k when he/she has access to the value of the observable variable x via power side-channels. The first-order probing security exactly characterizes the leakage model under the thread model introduced above.

Example 2.4. Consider the program P shown in Fig. 4. For any observable variable $x \in X^{\mathtt{main}_{in}}$, we can examine that $[\![\mathcal{E}(x)]\!]_\eta^P = \frac{1}{2}$ for any valuation η of the inputs $\{k_1, k_2\}$. It implies that x is $\emptyset$-simulatable, thus the program P first-order probing secure.

In contrast, $[\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_1}^P \neq [\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_2}^P$ if $\eta_1(k_1) \oplus \eta_1(k_2) = 0$ and $\eta_2(k_1) \oplus \eta_2(k_2) = 1$. Indeed, if $\eta_1(k_1) \oplus \eta_1(k_2) = 0$ and $\eta_2(k_1) \oplus \eta_2(k_2) = 1$, then

- $[\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_1}^P(0,0) = [\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_1}^P(1,1) = \frac{1}{2}$, and
- $[\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_2}^P(0,1) = [\![\vec{c}[1], \vec{c}[2]]\!]_{\eta_2}^P(1,0) = \frac{1}{2}$.

Thus, $\{\vec{c}[1], \vec{c}[2]\}$ is not $\emptyset$-simulatable. By fixing the inputs $\{k_1, k_2\}$, $[\![\vec{c}[1], \vec{c}[2]]\!]_\eta^P = [\![\vec{c}[1], \vec{c}[2]]\!]_\eta^P$, thus $\{\vec{c}[1], \vec{c}[2]\}$ becomes $\{k_1, k_2\}$-simulatable. □

2.3 A Running Example

Suppose one wants to compute $k_1 \odot (k_1 \oplus k_2)$, where k_1 and k_2 are two private inputs. It is known that the power consumption of computing both $k_1 \oplus k_2$ and $k_1 \odot (k_1 \oplus k_2)$ depends on the values of k_1 and k_2, based on which *differential power analysis* [72] could be utilized to infer the information of k_1 and k_2. For instance, $k_1 \oplus k_2$ is always 0 if $k_1 = k_2$ and is always 1 if $k_1 \neq k_2$. The static leakage current of a CMOS transistor depends on whether the transistor is on or off which corresponds to the logical 1 and 0 of a bit. This difference allows the adversary to infer if $k_1 = k_2$ or not by measuring the power consumption.

To ensure first-order probing security of computing $k_1 \odot (k_1 \oplus k_2)$, the distributions of internal variables for computing $k_1 \oplus k_2$ and $k_1 \odot (k_1 \oplus k_2)$ should be (statistically) independent of the values of k_1 and k_2. Namely, the distribution of static leakage currents of each CMOS transistor is independent of the values of k_1 and k_2.

```
1  main(k_1, k_2){
2    a⃗ = Encoding(k_1);
3    b⃗ = Encoding(k_2);
4    c⃗ = XORMULTI(a⃗, b⃗);}
5   // ⨁ c⃗ = k_1 ⊙ (k_1 ⊕ k_2)
6
7  XORMULTI(a⃗, b⃗){
8    e⃗ = Refresh(a⃗);
9    c⃗ = XOR(b⃗, e⃗);
10   d⃗ = UMA(e⃗, c⃗);
11   return d⃗;}
12  // ⨁ d⃗ = (⨁ a⃗) ⊙ (⨁ a⃗ ⊕ ⨁ b⃗)

13 Encoding(k){
14   a⃗[1] = $;
15   a⃗[2] = a⃗[1] ⊕ k;
16   return a⃗;} // ⨁ a⃗ = k
17 Refresh(a⃗){
18   r_1 = $;
19   c⃗[1] = a⃗[1] ⊕ r_1;
20   c⃗[2] = a⃗[2] ⊕ r_1;
21   return c⃗; } // ⨁ c⃗ = ⨁ a⃗
22 XOR(a⃗, b⃗){
23   for(i = 1; i ≤ 2; i++){
24     c⃗[i] = a⃗[i] ⊕ b⃗[i];}
25   return c⃗; } // ⨁ c⃗ = ⨁ a⃗ ⊕ ⨁ b⃗

26 UMA(a⃗, b⃗){ // Unified Multiplication
27   t_1 = a⃗[1] ⊙ b⃗[1];   t_2 = a⃗[2] ⊙ b⃗[2];   t_3 = a⃗[1] ⊙ b⃗[2];
28   t_4 = a⃗[2] ⊙ b⃗[1];   r_2 = $;
29   t_5 = t_1 ⊕ r_2;      // t_5 = (a⃗[1] ⊙ b⃗[1]) ⊕ r_2
30   t_6 = t_5 ⊕ t_3;      // t_6 = (a⃗[1] ⊙ b⃗[1]) ⊕ r_2 ⊕ (a⃗[1] ⊙ b⃗[2])
31   t_7 = t_2 ⊕ r_2;      // t_7 = (a⃗[2] ⊙ b⃗[2]) ⊕ r_2
32   t_8 = t_7 ⊕ t_4;      // t_8 = (a⃗[2] ⊙ b⃗[2]) ⊕ r_2 ⊕ (a⃗[2] ⊙ b⃗[1])
33   return (t_6, t_8);}   // t_6 ⊕ t_8 = (⨁ a⃗) ⊙ (⨁ b⃗)
```

Fig. 5. A running example for computing $k_1 \odot (k_1 \oplus k_2)$

Fig. 5 shows a demonstrating masked implementation for computing $k_1 \odot (k_1 \oplus k_2)$ using Boolean masking. Given the inputs k_1 and k_2, their encodings $\vec{a}$ and $\vec{b}$ are computed by calling the encoding gadget `Encoding` with the inputs k_1 and k_2 respectively (Lines 2–3). The `Encoding` gadget splits an input k into two shares $\vec{a}[1]$ and $\vec{a}[2]$ where $\vec{a}[1]$ is randomly sampled and $\vec{a}[2] = \vec{a}[1] \oplus k$ masks k by XORing $\vec{a}[1]$ resulting in the encoding $\vec{a}$ of k.

Based on the encodings $\vec{a}$ and $\vec{b}$, the encoding $\vec{c}$ of $k_1 \odot (k_1 \oplus k_2)$ is computed via invoking the composite gadget `XORMULTI` (Line 4). The value of $k_1 \odot (k_1 \oplus k_2)$ can be recovered by demasking, i.e. $\bigoplus \vec{c} = \vec{c}[1] \oplus \vec{c}[2] = k_1 \odot (k_1 \oplus k_2)$. Three simple gadgets `Refresh` [63], `XOR` [63] and `UMA` [58] are invoked in the gadget `XORMULTI`. The gadget `Refresh` takes an encoding as input and re-masks the shares via a fresh random variable. `Refresh` is required when the same random variable is used into two different sub-expressions in the same computation, e.g., $\vec{a}[2] \odot \vec{a}[1] = (\vec{a}[1] \oplus k_1) \odot \vec{a}[1]$ depends on the private variable k_1. The gadget `XOR` performs share-wise XOR, i.e., produces the encoding $(\vec{a}[1] \oplus \vec{b}[1], \vec{a}[2] \oplus \vec{b}[2])$ for the input encodings $\vec{a}$ and $\vec{b}$. The gadget `UMA` implements the finite-filed multiplication of the input encodings $\vec{a}$ and $\vec{b}$, so the output encoding (t_6, t_8) satisfying $t_6 \oplus t_8 = (\bigoplus \vec{a}) \odot (\bigoplus \vec{b})$.

Consider the variable $\vec{c}[1]@8@4$ which is the inlined version of the local variable $\vec{c}[1]$ in the gadget `Refresh` after inlining the gadget call to `Refresh` at Line 8 and the gadget call to the gadget `XORMULTI` at Line 4 respectively. Obviously, $\mathcal{E}(\vec{c}[1]@8@4) = (\vec{a}[1]@2 \oplus k_1) \oplus r_1@8@4$. Since k_1 in $\mathcal{E}(\vec{c}[1]@8@4)$ is perfectly masked by the random variable $r_1@8@4$ (resp. $\vec{a}[1]@2$), we can get that $[\![\mathcal{E}(\vec{c}[1]@8@4)]\!]^P_{\eta_1} = [\![\mathcal{E}(\vec{c}[1]@8@4)]\!]^P_{\eta_2}$ for any valuations $\eta_1, \eta_2 : \{k_1, k_2\} \to \mathbb{F}$. Thus, $\vec{c}[1]@8@4$ is $\emptyset$-simulatable. Indeed, we can observe that the local variable $\vec{c}[1]$ in the simple gadget `Refresh` is perfectly masked by the random variable r_1 without considering the calling context. It means that $\vec{c}[1]$ is always independent of private inputs no matter how the gadget `Refresh` is

invoked. Similarly, consider the local variable t_5 in the gadget UMA. Since $t_5 = (\vec{a}[1] \odot \vec{b}[1]) \oplus r_2$, we can deduce that t_5 is always independent of private inputs due to the random variable r_2 and operation $\oplus$ no matter how the gadget UMA is invoked.

Outline of the solution. To automatically prove that masked implementations of cryptographic algorithms such as the running example are first-order probing secure without inlining gadget calls, we shall propose a compositional verification approach by leveraging *local* random variables. We first introduce the new security notion $\mathbb{I}$-Non-Interference in terms of simulatability, that is parameterized by an explicit pre-condition $\mathbb{I}$ for each gadget (Section 3.1). The pre-condition of each gadget is served as an abstraction of the gadget when reasoning about the calls to the gadget so that the inlining of the gadget calls can be avoided. Then, we propose a composition rule for inferring pre-condition $\mathbb{I}$ of composite gadgets using the pre-conditions of called gadgets without inlining gadget calls (Section 3.2). The new security notion and its composition rule lay the foundation of our compositional verification approach. To reduce verification time, we further propose the concepts of dominant and dominated variables (Section 4.1), present a sound approach for determining them (Section 4.2) and show how to use them to reduce the size of pre-condition $\mathbb{I}$ for a composite gadget by revising the composition rule. Finally, after introducing a proof system for checking simulatability (Section 5.1), we present algorithms to infer pre-conditions for simple gadgets (Section 5.2) and composite gadgets (Section 5.3), respectively, by utilizing the proposed proof system and composition rules.

3 FOUNDATION OF OUR COMPOSITIONAL VERIFICATION APPROACH

In this section, we first introduce the new security notion $\mathbb{I}$-Non-Interference in terms of simulatability, parameterized by an explicit pre-condition $\mathbb{I}$ for each gadget, and then discuss how to make composition based on $\mathbb{I}$-Non-Interference by presenting a compositional rule.

3.1 $\mathbb{I}$-Non-Interference

To achieve compositional verification for masked implementations of cryptographic algorithms, we propose the notion of $\mathbb{I}$-Non-Interference. Let $\mathcal{P}(\cdot)$ denote the power set of a set.

Definition 3.1. Given a set of variable sets $\mathbb{I} \subseteq \mathcal{P}(X^{g_{in}} \cup X_a^g)$ of a gadget g, the gadget g is *$\mathbb{I}$-Non-Interfering* ($\mathbb{I}$-NI for short), if for every variable $x \in X^{g_{in}} \cup X_a^g$, there exists a variable set $I \in \mathbb{I}$ such that x is I-simulatable. (Note that g_{in} is the inlined version of the gadget g and $X^{g_{in}}$ is the set of internal variables defined in the inlined version g_{in}.)

The pre-condition $\mathbb{I}$ in the notion $\mathbb{I}$-NI characterizes sufficient variable sets to simulate the distributions of all observable variables. The advantage of the pre-condition $\mathbb{I}$ is twofold. First, it can contain as few small variable sets as possible to simulate all the observable variables in a gadget. Second, if an observable variable x cannot be simulated by any variable set into $\mathbb{I}$, the variable x could be added to the set $\mathbb{I}$ to ensure that the gadget is $\mathbb{I}$-NI. The new notion $\mathbb{I}$-NI can be seen as a balance of the first-order probing security and the stronger security notion proposed by Barthe et al. [9], where the former does not support compositional verification, while the latter implicitly contains a fixed pre-condition. We note that the fixed pre-condition may not be fulfilled by generic, in particular, efficient, masked implementations though it is more composition-friendly. More importantly, it is straightforward to check first-order probing security from our notion $\mathbb{I}$-NI by the following proposition.

Proposition 3.2. *If a gadget g is both $\mathbb{I}$-NI and I is X_p-simulatable for each variable set $I \in \mathbb{I}$, then g is first-order probing secure, namely, any variable of the gadget g is statistically independent of the private inputs X_k when g is used in a program P with the private inputs X_k.*

Proof. Suppose the gadget g is both $\mathbb{I}$-NI and I is X_p-simulatable for each variable set $I \in \mathbb{I}$. To show that g is first-order probing secure, it suffices to prove that for any program P that uses the gadget g and any observable variable $x \in X^{g_{in}} \cup X^g_a$, $x@$ is X_p-simulatable, i.e., any inlined version $x@$ of x in the program P_{in} is statistically independent of the private inputs X_k.

Consider an observable variable $x \in X^{g_{in}} \cup X^g_a$. Since the gadget g is $\mathbb{I}$-NI, there must exist a variable set $I \in \mathbb{I}$ such that x is I-simulatable. By Proposition 2.3, we get that for any inlined $x@$ version of the variable x from the gadget g in the program P_{in}, $x@$ is X_p-simulatable. □

Example 3.3. Consider the running example. Let $\mathbb{I}_{\texttt{Refresh}}$, $\mathbb{I}_{\texttt{XOR}}$ and $\mathbb{I}_{\texttt{UMA}}$ be the following sets:

- $\mathbb{I}_{\texttt{Refresh}} = \{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}$;
- $\mathbb{I}_{\texttt{XOR}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}$;
- $\mathbb{I}_{\texttt{UMA}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}, \{\vec{a}[1], \vec{b}[2]\}, \{\vec{a}[2], \vec{b}[1]\}\}$.

It is easy to verify that:

- In Refresh, r_1 and $\vec{c}[1]$ and $\vec{c}[2]$ are $\emptyset$-simulatable, and $\vec{a}[i]$ is $\vec{a}[i]$-simulatable for $i = 1, 2$;
- In XOR, $\vec{a}[i]$ is $\vec{a}[i]$-simulatable, $\vec{b}[i]$ is $\vec{b}[i]$-simulatable, and $\vec{c}[i]$ is $\{\vec{a}[i], \vec{b}[i]\}$-simulatable for $i = 1, 2$;
- In UMA, $\vec{a}[i]$ is $\vec{a}[i]$-simulatable, $\vec{b}[i]$ is $\vec{b}[i]$-simulatable, $\vec{t}[i]$ is $\{\vec{a}[i], \vec{b}[i]\}$-simulatable for $i = 1, 2$, t_3 is $\{\vec{a}[1], \vec{b}[2]\}$-simulatable, t_4 is $\{\vec{a}[2], \vec{b}[1]\}$-simulatable, all of r_2, t_5, t_6, t_7 and t_8 are $\emptyset$-simulatable.

Thus, Refresh is $\mathbb{I}_{\texttt{Refresh}}$-NI, XOR is $\mathbb{I}_{\texttt{XOR}}$-NI and UMA is $\mathbb{I}_{\texttt{UMA}}$-NI. □

In Section 5.2, we will present an algorithm for inferring pre-conditions of simple gadgets. An algorithm for inferring pre-conditions of composite gadgets will be given in Section 5.3 which relies upon pre-conditions of simple gadgets and the composition rule introduced below.

3.2 Composition of $\mathbb{I}$-NI Gadgets

We show the simplicity of inferring pre-conditions for composite gadgets by proposing a composition rule. We use the following composite gadget to illustrate the composition rule,

$$f(\vec{x}_1, \cdots, \vec{x}_m)\{\ \vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m);\ \vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n);\ \texttt{return}\ \vec{z};\ \}$$

where f is a composite gadget that contains two gadget call statements to the simple gadgets g and h (with labels ℓ_g and ℓ_h) respectively. The actual parameters of g are the formal parameters of f, and the actual parameters of h consist of the return values of g and formal parameters of f. Assume the gadget $g(\vec{a}_1, \cdots, \vec{a}_m)\{\cdots\}$ is $\mathbb{I}_g$-NI and the gadget $h(\vec{b}_1, \cdots, \vec{b}_n)\{\cdots\}$ is $\mathbb{I}_h$-NI. We denote by $\mathbb{I}_g[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g$ the set $\mathbb{I}_g$ after instantiating the formal parameters $\vec{a}_i$'s by the corresponding actual parameters $\vec{x}_i$'s and local variables are appended with $@\ell_g$ to avoid name conflict. The set $\mathbb{I}_h[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$ is defined similarly. Let $\mathbb{I}_f$ be the following set

$$\mathbb{I}_g[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \cup \mathbb{I}_h[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h.$$

Proposition 3.4. *The gadget f is $\mathbb{I}_f$-NI.*

Proof. To prove that the gadget f is $\mathbb{I}_f$-NI, it suffices to prove that for every observable variable x of the inlined version f_{in}, the variable x in f_{in} is I-simulatable for some variable set $I \in \mathbb{I}_f$. We first consider inlined versions of the variables from the gadgets h and f, then move on to the observable variables defined in the gadget f.

- Consider an inlined version $x@\ell_g$ of a variable x from the gadget g. Since the gadget g is $\mathbb{I}_g$-NI, there exists a variable set $I \in \mathbb{I}_g$ such that the variable x in g is I-simulatable. It implies

that the variable $x@\ell_g$ in f_{in} is $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g$-simulatable. The result immediately follows from the fact that $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \in \mathbb{I}_g[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \subseteq \mathbb{I}_f$.

- Consider an inlined version $x@\ell_h$ of a variable x from the gadget h, similar to the above case, we can get that the variable $x@\ell_h$ in f_{in} is $I[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$-simulatable for some set $I \in \mathbb{I}_h$ and $I[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h \in \mathbb{I}_h[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h \subseteq \mathbb{I}_f$.
- Consider an observable variable x defined in the gadget f. Then, x must be an actual parameter or a return value of one of the gadget call statements $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$.

  If x is an actual parameter of $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$, let c be the corresponding formal parameter of x. Then, the variable c in the gadget g is I-simulatable for some $I \in \mathbb{I}_g$ or the variable c in the gadget h is I-simulatable for some $I \in \mathbb{I}_h$. Since x and c always have the same value, we get that the variable x in f_{in} is I'-simulatable, where I' is $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g$ or $I[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$.

  If x is a return value of $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$, let c be the corresponding return variable of x in the gadget g or h. We have proved that $c@\ell_g$ (resp. $c@\ell_h$) is I-simulatable if c is a return variable of g (resp. h) for some $I \in \mathbb{I}_f$. Since x and c always have the same value, we get that the variable x in the gadget f_{in} is I-simulatable as well.

We have proved that every observable variable x of the inlined version f_{in} is I-simulatable, for some variable set $I \in \mathbb{I}_f$, thus conclude the proof. □

Example 3.5. We here show how to manually compute the pre-condition $\mathbb{I}_{\texttt{XORMULTI}}$ such that the composite gadget XORMULTI in the running example is $\mathbb{I}_{\texttt{XORMULTI}}$-NI by leveraging this composition rule. (An algorithm for inferring pre-conditions of composite gadgets will be given in Section 5.3.) Recall that Refresh is $\mathbb{I}_{\texttt{Refresh}}$-NI, the gadget XOR is $\mathbb{I}_{\texttt{XOR}}$-NI and the gadget UMA is $\mathbb{I}_{\texttt{UMA}}$-NI (cf. Example 3.3), where

- $\mathbb{I}_{\texttt{Refresh}} = \{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}$;
- $\mathbb{I}_{\texttt{XOR}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}$;
- $\mathbb{I}_{\texttt{UMA}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}, \{\vec{a}[1], \vec{b}[2]\}, \{\vec{a}[2], \vec{b}[1]\}\}$.

The set $\mathbb{I}_{\texttt{XORMULTI}}$ is computed as follows according to the composition rule:

(1) For the gadget call $\vec{e} = \texttt{Refresh}(\vec{a})$, we have:
$$\mathbb{I}_{\texttt{Refresh}}@8 = \mathbb{I}_{\texttt{Refresh}}[\vec{a}/\vec{a}]@8 = \{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}.$$
(2) For the gadget call $\vec{c} = \texttt{XOR}(\vec{b}, \vec{e})$, we have:
$$\mathbb{I}_{\texttt{XOR}}@9 = \mathbb{I}_{\texttt{XOR}}[\vec{b}/\vec{a}, \vec{e}/\vec{b}]@9 = \{\{\vec{b}[1], \vec{e}[1]\}, \{\vec{b}[2], \vec{e}[2]\}\}.$$
(3) For the gadget call $\vec{d} = \texttt{UMA}(\vec{e}, \vec{c})$, we have:
$$\mathbb{I}_{\texttt{UMA}}@10 = \mathbb{I}_{\texttt{UMA}}[\vec{e}/\vec{a}, \vec{c}/\vec{b}]@10 = \{\{\vec{e}[i], \vec{c}[j]\} | 1 \leq 1, j \leq 2\}.$$

Let $\mathbb{I}_{\texttt{XORMULTI}} = \mathbb{I}_{\texttt{Refresh}}@8 \cup \mathbb{I}_{\texttt{XOR}}@9 \cup \mathbb{I}_{\texttt{UMA}}@10 = \{\{\vec{a}[i]\}, \{\vec{b}[i], \vec{e}[i]\}, \{\vec{e}[i], \vec{c}[j]\} | 1 \leq i, j \leq 2\}$. By Proposition 3.4, XORMULTI is $\mathbb{I}_{\texttt{XORMULTI}}$-NI. □

4 DOMINANT AND DOMINATED VARIABLES

As mentioned previously, the pre-condition $\mathbb{I}$ of a gadget is expected to contain as few small variable sets as possible to simulate all observable variables. However, directly applying the composition rule may yield a pre-condition containing many large variable sets. To alleviate this issue, in this section, we first introduce dominant and dominated variables, then provide an approach to identify them, and finally show how such information can help further reduce the size of pre-condition $\mathbb{I}$ in the composition of gadgets by improving the composition rule.

4.1 The Concepts of Dominant and Dominated Variables

Given a variable x, let $\mathsf{Sub}(x)$ be the set of the sub-expressions of $\mathcal{E}(x)$.

Definition 4.1. A variable x is dominated by another variable y if y occurs in $\mathcal{E}(x)$ only once and each operation $\circ$ along the path from y to the root of the abstract syntax tree of $\mathcal{E}(x)$ is either from or belongs to following:

- $\{\oplus, +, -, \neg\}$;
- $\odot$ and one of its children is non-zero constant,

where x is called a **dominated variable** and y is called a **dominant variable** of x.

An encoding $\vec{x}$ is dominated by another encoding $\vec{y}$ if each share $\vec{x}[i]$ of the encoding $\vec{x}$ is dominated by only one share $\vec{y}[j]$ of the encoding $\vec{y}$, where $\vec{x}$ is called **dominated encoding** and $\vec{y}$ is called **dominant encoding** of $\vec{x}$.

Intuitively, if the variable x is dominated by the another variable y, the distribution of (the value of) of x is determined by the distribution of y. When y is a random variable, the distribution of x is uniform as well, thus x can be regarded as a random variable. For example, consider $y = x \oplus r$, $y' = x \wedge r$ and $y'' = x + (x \oplus r)$. The variable y is dominated by both x and r because of the logical operation $\oplus$ and the uniqueness of x and r, but y' is dominated by neither x nor r due to the logical operation $\wedge$, and y'' is only dominated by r because x occurs twice. When r is a random variable, the distributions of y and y'' are uniform for any fixed value of x.

Let $\mathsf{DomR}(x)$ be the set of random dominant variables of the variable x. The merit of dominated variables is justified by the following straightforward proposition.

Proposition 4.2. *x has a uniform distribution if* $\mathsf{DomR}(x) \neq \emptyset$.

Proof. Suppose $r \in \mathsf{DomR}(x)$. We first prove the following claim.

> **Claim**. Let e be a sub-expression of $\mathcal{E}(x)$ such that r occurs in e and $\mathcal{E}(x)[r/e]$ be the expression $\mathcal{E}(x)$ in which all the occurrences of the sub-expression e are replaced by the random variable r. $\mathcal{E}(x)$ and $\mathcal{E}(x)[r/e]$ have the same distribution if e is in the form of $e' \circ r$ for $\circ \in \{\oplus, +, -\}$ or $\neg r$ or $c \odot r$ such that c is non-zero constant.

Proof of the claim. Since $r \in \mathsf{DomR}(x)$, the random variable r occurs only once in $\mathcal{E}(x)$. It means that the random variable r occurs only once in e, but does not appear in e'. Since r is uniformly sampled from $\mathbb{F}$, we get that the value of e is determined by the value of r for any fixed value of e' or c. Let $f_v(r)$ be the function such that the value of e' or c is fixed to v. Obviously, $f_v(r)$ is a bijective function. Thus, the probability distribution of $f_v(r)$ is the same as r, implying that the probability distribution of the value of e is uniform when r is uniformly sampled from $\mathbb{F}$. Since r does not appear anywhere in $\mathcal{E}(x)$ except for the sub-expression e, we get that $\mathcal{E}(x)$ and $\mathcal{E}(x)[r/e]$ have the same distribution. We conclude the proof of the claim.

To show that x has a uniform distribution, we can iteratively replace all the occurrences of the sub-expression e in $\mathcal{E}(x)$ by the random variable r for every sub-expression e in the form of $e' \circ r$ for $\circ \in \{\oplus, +, -\}$ or $\neg r$ or $c \odot r$ such that c is non-zero constant. The expression $\mathcal{E}(x)$ will eventually becomes the random variable r according to Definition 4.1. By the above lemma, these substitutions do not change the probability distribution of $\mathcal{E}(x)$. Thus, x has a uniform distribution. □

Example 4.3. Consider the gadget `Refresh` in the running example. Since $\mathcal{E}(\vec{c}[1]) = \vec{a}[1] \oplus r_1$ and $\mathcal{E}(\vec{c}[2]) = \vec{a}[2] \oplus r_1$, we can deduce that $\vec{c}[1]$ is a dominated variable and dominated by both $\vec{a}[1]$ and r_1, and the variables $\vec{a}[1]$ and r_1 are dominant variable of $\vec{c}[1]$. Similarly, $\vec{c}[2]$ is dominated by both $\vec{a}[2]$ and r_1, and the variables $\vec{a}[2]$ and r_1 are dominant variable of $\vec{c}[2]$. Since r_1 is a random

variable, we get that $\mathsf{DomR}(\vec{c}[1]) = \mathsf{DomR}(\vec{c}[2]) = \{r_1\}$, and both $\vec{c}[1]$ and $\vec{c}[2]$ have a uniform distribution for any inputs of the gadget `Refresh`.

Since each share $\vec{c}[i]$ of the encoding $\vec{c}$ is dominated by only one share $\vec{a}[i]$ of the encoding $\vec{a}$, we get that the encoding $\vec{c}$ is a dominated encoding and dominated by the encoding $\vec{a}$, and the encoding $\vec{a}$ is a dominant encoding of the encoding $\vec{c}$.

In contrast, $\vec{c}[1]$ is dominated by neither $\vec{a}[1]$ nor r_1 if $\mathcal{E}(\vec{c}[1])$ is $\vec{a}[1] \wedge r_1$ (because of $\wedge$) or $(\vec{a}[1] \wedge r_1) \oplus r_1$ (because of twice occurrences of r_1 and $\wedge$). Thus, $\mathsf{DomR}(\vec{c}[1]) = \emptyset$ and $\vec{c}[1]$ does not necessarily have a uniform distribution although r_1 is a random variable.

If $\mathcal{E}(\vec{c}[1])$ is $(\vec{a}[1] \vee \vec{a}[2]) \oplus (\neg r_1)$, $\vec{c}[1]$ is only dominated by the variable r_1 (because of $\vee$), then the encoding $\vec{c}$ is not dominated by the encoding $\vec{a}$ because the share $\vec{c}[1]$ is not dominated by any share of $\vec{a}$. Moreover, $\mathsf{DomR}(\vec{c}[1]) = \{r_1\}$ and $\vec{c}[1]$ has a uniform distribution as r_1 is a random variable.

If $\mathcal{E}(\vec{c}[1])$ is $(\vec{a}[1] \oplus \vec{a}[2]) \oplus r_1$, $\vec{c}[1]$ is dominated by $\vec{a}[1]$, $\vec{a}[2]$ and r_1, then the encoding $\vec{c}$ is not dominated by the encoding $\vec{a}$ because the share $\vec{c}[1]$ of the encoding $\vec{c}$ is dominated by both shares $\vec{a}[1]$ and $\vec{a}[2]$ of the encoding $\vec{a}$. Similarly, $\mathsf{DomR}(\vec{c}[1]) = \{r_1\}$ and $\vec{c}[1]$ has a uniform distribution. □

Proposition 4.2 provides a sufficient condition to deduce that x has the uniform distribution, so can be simulated by an empty set, i.e., x is $\emptyset$-simulatable. We will also show that variables dominated by random variables can help simplify expressions later (cf. Section 5).

4.2 Determining Dominated Variables

In this section, we present an efficient approach to determine dominant and dominated variables. We first consider simple gadgets and then composite gadgets.

4.2.1 Determining Dominated Variables for Simple Gadgets. Determining dominated variables in simple gadgets is trivial. For every variable x of a simple gadget, we represent the computation $\mathcal{E}(x)$ of the variable x as a Directed Acyclic Graph (DAG), where the internal nodes are labeled by operators and leave are labeled by variables and constants involved in the computation $\mathcal{E}(x)$. We iteratively traverse the DAG representation of $\mathcal{E}(x)$ starting from the root in a depth-first fashion, where a stack is used to store visiting nodes. For each node in the DAG,

(1) if the in-degree of the node is greater than 1, then we go back to its visiting parent (popped from the stack) if exists otherwise terminate, because any variable labeled to the leave of the sub-tree rooted by this node occurs at least twice in the computation $\mathcal{E}(x)$;
(2) if the in-degree of the node is 1, we proceed as follows:
   (a) if the node is labeled by one of the operators $\{\oplus, +, -, \neg\}$, we push it into the stack and continue to traverse its unvisited children;
   (b) if the node is labeled by the operator $\odot$, one child of the node is a non-zero constant and the another child is a non-constant that has not been visited yet, we push it into the stack and continue to traverse its non-constant child;
   (c) otherwise we go back to its visiting parent if exists otherwise terminate, because any variable labeled to the leave of the sub-tree rooted by this node cannot be a dominant variable of x according to Definition 4.1;
(3) if the node is a leaf, with in-degree no more than 1, and labeled by a variable, then the variable must be a dominant variable of x, we record this dominant variable and go back to its visiting parent.

By applying the above procedure, we can identify all the dominant variables of the variable x in linear time in the size of the computation $\mathcal{E}(x)$. Similarly, the set $\mathsf{DomR}(x)$ of random dominant variables of the variable x can be computed.

4.2.2 Determining Dominated Variables for Composite Gadgets. It is non-trivial to determine if variables (encodings) are dominated by random variables in composite gadgets without inlining gadget calls, as they are return values of gadget calls. To address this issue, we first characterize whether a gadget can transfer dominant variables from a formal parameter to its output encoding and whether a gadget can generate dominant variables itself, based on which we introduce the concept of summary for storing these information. Next, we present an approach for under-approximating the summaries of composite gadgets by utilizing the summaries of called gadgets.

Characterzation of gadgets. The characterzation of gadgets is inspired by following observation. First, the return of a gadget call statement may be dominated by the same random variables as some actual parameters of the gadget call statement, i.e., random variables may be transferred from the input encodings to the output encoding, making the output encoding being dominated by the same dominant encodings of some input encodings. Second, the output encoding of a gadget may be dominated by local random variables of the gadget. Based on these observations, we characterize the gadgets that can transfer dominant variables and generate dominant variables as follows.

Fix a gadget $g(\vec{a}_1, \cdots, \vec{a}_m)\{\cdots; \texttt{return}\ \vec{o};\}$.

Definition 4.4. The gadget g can *transfer dominant variables* from a formal parameter $\vec{a}_i$ to its output encoding $\vec{o}$ if $\vec{a}_i$ is a dominant encoding of $\vec{o}$.

Definition 4.5. The gadget g can *generate dominant variables* if each share $\vec{o}[j]$ of the output encoding $\vec{o}$ is dominated by a local random variable of g.

Summary of gadgets. We define the *summary* of the gadget g as $T_g \subseteq X^g_{en} \cup \{\vec{o}\}$, where T_g includes all the formal parameters $\vec{a}_i$ of the gadget g whose dominant encodings can be transferred to the output encoding $\vec{o}$, and the output encoding $\vec{o}$ is added to the summary T_g if the gadget g itself can generate dominant variables. Based on Definition 4.4 and Definition 4.5, the summary T_g can be easily computed if g is a simple gadget. More specifically, for each share $\vec{o}[j]$ of the output encoding $\vec{o}$, we first compute the dominant variables from the computation $\mathcal{E}(\vec{o}[j])$, then add an input encoding $\vec{a}_i$ into the summary T_g if each share $\vec{o}[j]$ of the output encoding $\vec{o}$ is dominated by only one share of the input encoding $\vec{a}_i$, and finally the output encoding $\vec{o}$ is added into the summary T_g if each share $\vec{o}[j]$ of the output encoding $\vec{o}$ is dominated by a random variable.

Example 4.6. Consider the gadget `Refresh` in the running example. Since $\vec{c}[1] = \vec{a}[1] \oplus r_1$ and $\vec{c}[2] = \vec{a}[2] \oplus r_1$ with random variables r_1 and r_2, `Refresh` can generate dominant variables (i.e., r_1 and r_2), thus the output encoding $\vec{c}$ of the gadget `Refresh` can be added into the summary $T_{\texttt{Refresh}}$ of the gadget `Refresh`, i.e., $\vec{c} \in T_{\texttt{Refresh}}$. Since the output encoding $\vec{c}$ of the gadget `Refresh` is dominated by the input encoding $\vec{a}$, the input encoding $\vec{a}$ can also be added into the summary $T_{\texttt{Refresh}}$ of the gadget `Refresh`, i.e., $\vec{a} \in T_{\texttt{Refresh}}$. Thus, we have $T_{\texttt{Refresh}} = \{\vec{a}, \vec{c}\}$. Similarly, we can deduce that $T_{\texttt{XOR}} = \{\vec{a}, \vec{b}\}$ and $T_{\texttt{UMA}} = \{(t_6, t_8)\}$. □

Under-approximating the summary of composite gadgets. Let E be the set of all the encodings of the gadget g. To compute the summary T_g of the gadget g without inlining gadget calls when g is a composite gadget, we define a function $\lambda : E \to \mathcal{P}(E)$ such that for each encoding $\vec{a} \in E$, $\lambda(\vec{a})$ contains all the dominant encodings of the encoding $\vec{a}$, and moreover $\vec{a} \in \lambda(\vec{a})$ *only* if $\vec{a}$ is the return of a gadget call that can generate dominant variables. Thus, it is easy to see that:

Proposition 4.7. *$\lambda(\vec{o}) \cap (X^g_{en} \cup \{\vec{o}\}) \subseteq T_g$, namely, the summary T_g of the composite gadget can be under-approximated by computing the function λ.*

Proof. According to the definition of the function λ, all the dominant encodings of the output encoding $\vec{o}$ are contained in $\lambda(\vec{o})$. Thus, the intersection of $\lambda(\vec{o})$ and X^g_{en} is a set of dominant encodings of the output encoding $\vec{o}$ that are formal parameters of the gadget g. Furthermore $\vec{o} \in \lambda(\vec{o})$ *only* if $\vec{o}$ is the return of a gadget call that can generate dominant variables, thus the gadget g itself can generate dominant variables. The result follows from the definition of T_g. □

Note that X^g_{en}, the vector of all the input encodings of the gadget g, is used as a set of input encodings in the definition of T_g and Proposition 4.7.

Now the problem is how to compute λ. To achieve this, we first show how to compute the dominant encodings $\lambda(\vec{y})$ of an internal encoding $\vec{y}$ defined by $\vec{y} = f(\vec{x}_1, \cdots, \vec{x}_k)$, based on the summary T_f of the gadget f and dominant encodings of the actual parameters, i.e. $\lambda(\vec{x}_1), \cdots, \lambda(\vec{x}_k)$. We assume that the formal parameters of the gadget f are $\vec{b}_1, \cdots, \vec{b}_k$. First, we can directly get:

Proposition 4.8. *If the gadget f can generate dominant variables, then $\vec{y}$ can be added into $\lambda(\vec{y})$.*

Proof. Let $\lambda(\vec{o})$ be the output encoding of the gadget f. Suppose the gadget f can generate dominant variables, then $\vec{o} \in \lambda(\vec{o})$. Since $\vec{y} = f(\vec{x}_1, \cdots, \vec{x}_k)$, i.e., $\vec{y}$ is the return of a gadget call that can generate dominant variables, we can add $\vec{y}$ into $\lambda(\vec{y})$. □

Example 4.9. Consider the gadget `Refresh` in the running example. It can generate dominant variables (i.e., $\vec{c} \in T_{\texttt{Refresh}}$, cf. Example 4.6), thus $\vec{c} \in \lambda(\vec{c})$. Now, consider the gadget `XORMULTI` in the running example which has a gadget call to the gadget `Refresh`, i.e., $\vec{e} = \texttt{Refresh}(\vec{a})$. Thus, the encoding $\vec{e}$ can be added into $\lambda(\vec{e})$.

In contrast, the gadget `XOR` in the running example cannot generate dominant variables (i.e., $\vec{c} \notin T_{\texttt{XOR}}$, cf. Example 4.6). Thus, we cannot add the encoding $\vec{c}$ into $\lambda(\vec{c})$ for the gadget call statement $\vec{c} = \texttt{XOR}(\vec{b}, \vec{e})$. □

Hereafter, by mutually independence of the encodings $\vec{x}_1, \cdots, \vec{x}_k$ we mean that the distributions of all the encodings $\vec{x}_1, \cdots, \vec{x}_k$ are mutually independent. Note that distributions of the shares in the same encoding can be dependent. For instance, the encodings $\vec{a}$ and $\vec{b}$ in the `main` gadget in the running example are mutually independent, and the encodings $\vec{b}$ and $\vec{e}$ in the `XORMULTI` gadget in the running example are mutually independent as well, while the encodings $(r, r \oplus k_1)$ and $(r, r \oplus k_2)$ are not mutually independent even if r is a random variable. We have:

Proposition 4.10. *If the gadget f can transfer dominant variables from the formal parameter $\vec{b}_i$ to its output encoding (i.e., $\vec{b}_i \in T_f$) and the encodings $\vec{x}_1, \cdots, \vec{x}_k$ in $\vec{y} = f(\vec{x}_1, \cdots, \vec{x}_k)$ are mutually independent, then $\vec{y}$ is dominated by the dominant encodings of $\vec{x}_i$, namely, $\lambda(\vec{x}_i) \subseteq \lambda(\vec{y})$.*

Proof. Suppose the encoding $\vec{x}_i$ is dominated by an encoding $\vec{b}$ and the encodings $\vec{x}_1, \cdots, \vec{x}_k$ are mutually independent. Since the gadget f can transfer dominant variables from $\vec{b}_i$ to its output encoding $\vec{o}$ (i.e., $\vec{b}_i \in \lambda(\vec{o})$ or $\vec{b}_i \in T_f$), and $\vec{y} = f(\vec{x}_1, \cdots, \vec{x}_k)$, we get that $\vec{y}$ is dominated by the encoding $\vec{b}$, because the values of the encodings $\vec{x}_i$ and $\vec{b}_i$ (resp. $\vec{o}$ and $\vec{y}$) are the same. □

Remark that it is necessary to assume that the actual parameters are mutually independent in Proposition 4.10, because if shares between different actual parameters are not mutually independent, the computation of two such shares may not contain any dominant variables.

Example 4.11. Consider the gadget XOR in the running example. It can transfer dominant variables from the formal parameters $\vec{a}$ and $\vec{b}$ to the output encoding $\vec{c}$ (i.e., $T_{\texttt{XOR}} = \{\vec{a}, \vec{b}\}$, cf. Example 4.6). For the gadget call statement $\vec{c} = \texttt{XOR}(\vec{b}, \vec{e})$ in the gadget XORMULTI, the encodings $\vec{b}$ and $\vec{e}$ are mutually independent, thus we have: $\lambda(\vec{b}) \subseteq \lambda(c)$ and $\lambda(\vec{e}) \subseteq \lambda(c)$.

In contrast, for the gadget call $\vec{y} = \texttt{XOR}(\vec{x}_1, \vec{x}_2)$ such that $\vec{x}_1 = \vec{x}_2$, i.e., $\vec{x}_1$ and $\vec{x}_2$ are not mutually independent, $\vec{y}$ is not a dominated encoding because the shares $\vec{y}[1]$ and $\vec{y}[2]$ of $\vec{y}$ are 0 even though $T_{\texttt{XOR}} = \{\vec{a}, \vec{b}\}$. □

To decide whether the actual parameters in a gadget call are mutually independent, we propose a sufficient condition based on the function λ. Intuitively, for the gadget call $\vec{y} = \texttt{XOR}(\vec{x}_1, \vec{x}_2)$, if $\lambda(\vec{x}_1) = \{\vec{a}\}$, $\lambda(\vec{x}_2) = \{\vec{b}\}$, and $\vec{a}$ and $\vec{b}$ are mutually independent, then $\vec{x}_1$ and $\vec{x}_2$ are mutually independent as well. Formally, we have the following sufficient condition.

Sufficient condition. Let $\Psi(\vec{x}_1, \cdots, \vec{x}_k)$ be a predicate which holds only if $\lambda(\vec{x}_i) \neq \emptyset$ for each i and there exists a set of encodings from $\bigcup_{i=1}^{k} \lambda(\vec{x}_i)$, one encoding $\vec{z}_i$ per set $\lambda(\vec{x}_i)$, such that the encodings $\vec{z}_i$'s are mutually independent.

Proposition 4.12. *The encodings $\vec{x}_1, \cdots, \vec{x}_k$ are mutually independent if they are distinct and either $\Psi(\vec{x}_1, \cdots, \vec{x}_k)$ or $\vec{x}_1 \in \lambda(\vec{x}_1), \cdots, \vec{x}_k \in \lambda(\vec{x}_k)$.*

Proof. Suppose $\vec{x}_1, \cdots, \vec{x}_k$ are distinct encodings. On the one hand, if $\vec{x}_1 \in \lambda(\vec{x}_1), \cdots, \vec{x}_k \in \lambda(\vec{x}_k)$, then $\vec{x}_i$ must be the return of a gadget call that can generate dominant variables; meanwhile $\vec{x}_i$ and $\vec{x}_j$ are different returns. Thus, the distributions of $\vec{x}_1, \cdots, \vec{x}_k$ are mutually independent. On the other hand, if $\Psi(\vec{x}_1, \cdots, \vec{x}_k)$ holds, then the distributions of $\vec{x}_i$'s are the same as that of encodings $\vec{z}_i$'s. Thus, if $\vec{z}_i$'s are mutually independent, then $\vec{x}_1, \cdots, \vec{x}_k$ are mutually independent. □

By iteratively applying Propositions 4.8 and 4.10, a function λ can be computed from which we can under-approximate the summary T_g of a composite gadget g. Here, we demonstrate the general intuition on an example, as both the function λ and summary T_g are computed within the algorithm for inferring pre-conditions of composite gadgets (cf. Section 5.3).

Example 4.13. Let us consider the gadget XORMULTI in the running example. Initially, $\lambda(\vec{a}) = \lambda(\vec{b}) = \emptyset$.

(1) For the gadget call $\vec{e} = \texttt{Refresh}(\vec{a})$, since $\Psi(\vec{a})$ does not holds and $T_{\texttt{Refresh}} = \{\vec{a}, \vec{c}\}$ (cf. Example 4.6), we have $\lambda(\vec{e}) = \{\vec{e}\}$ according to Proposition 4.8 (cf. Example 4.9).
(2) For the gadget call $\vec{c} = \texttt{XOR}(\vec{b}, \vec{e})$, since $\Psi(\vec{b}, \vec{e})$ does not hold and $T_{\texttt{XOR}} = \{\vec{a}, \vec{b}\}$ (cf. Example 4.6), we have $\lambda(\vec{c}) = \emptyset$.
(3) For the gadget call $\vec{d} = \texttt{UMA}(\vec{e}, \vec{c})$, since $\lambda(\vec{e}) = \{\vec{e}\}$ and $\lambda(\vec{c}) = \emptyset$, $\Psi(\vec{e}, \vec{c})$ does not hold. As $T_{\texttt{UMA}} = \{(t_6, t_8)\}$ (cf. Example 4.6), we have $\lambda(\vec{d}) = \{\vec{d}\}$ according to Proposition 4.8.

Finally, we obtain that $T_{\texttt{XORMULTI}} = \lambda(\vec{d}) = \{\vec{d}\}$, namely, the gadget XORMULTI can generate dominant variables itself, and no dominant variables can be transferred from its parameters $\vec{a}$ and $\vec{b}$ to the output encoding. Note that in this case whether $\vec{a}$ and $\vec{b}$ are dominated encodings and whether they are mutually independent, are unknown, thus we set $\lambda(\vec{a}) = \lambda(\vec{b}) = \emptyset$. If the gadget XORMULTI is called in another gadget where $\lambda(\vec{a})$ and $\lambda(\vec{b})$ are nonempty, λ may have different values (cf. Example 4.15). □

4.3 Application of Dominated Variables

Dominated variables can help reduce the size of the pre-condition $\mathbb{I}$ when computing $\mathbb{I}$ for a composite gadget. More specifically, if the actual parameters of a gadget call have random dominant encodings, they have uniform distributions. Thus, when instantiating $\mathbb{I}$, the corresponding formal parameters can be instantiated by an empty set, rather than the actual parameters. Therefore, Proposition 3.4 can be refined as follows. Let $\mathbb{I}_f = \mathbb{I}_g[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \cup \mathbb{I}_h[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$ for the composite gadget f:

$$f(\vec{x}_1, \cdots, \vec{x}_m)\{\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m); \vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n); \texttt{return } \vec{z};\}$$

where the gadget $g(\vec{a}_1, \cdots, \vec{a}_m)\{\cdots\}$ is $\mathbb{I}_g$-NI and the gadget $h(\vec{b}_1, \cdots, \vec{b}_n)\{\cdots\}$ is $\mathbb{I}_h$-NI.

Proposition 4.14. *The gadget f is $\mathbb{I}_f$-NI, if there exists $\vec{x} \in \lambda(\vec{y}_1)$ such that $\vec{x} \in \lambda(\vec{x})$.*

Proof. The proof mainly follows the lines of the proof of Proposition 3.4.

Suppose $\vec{x} \in \lambda(\vec{y}_1)$ such that $\vec{x} \in \lambda(\vec{x})$. From $\vec{x} \in \lambda(\vec{x})$, we get that $\vec{x}$ is the return of a gadget call which can generate dominant variables itself, namely, each share $\vec{x}[j]$ of the encoding $\vec{x}$ is dominated by a random variable. By Proposition 4.2, each share $\vec{x}[j]$ of the encoding $\vec{x}$ has a uniform distribution. Since, $\vec{x} \in \lambda(\vec{y}_1)$, we get that each share $\vec{y}_1[j]$ of the encoding $\vec{y}_1$ is dominated by a random variable and has a uniform distribution.

To prove that the gadget f is $\mathbb{I}_f$-NI, it suffices to prove that every observable variable x of f_{in} is I-simulatable for some variable set $I \in \mathbb{I}_f$. We will start by examining the inlined versions of the variables from the gadgets g and h, and then move on to the observable variables defined in the gadget f.

- Consider an inlined version $x@\ell_g$ of a variable x from the gadget g. Since the gadget g is $\mathbb{I}_g$-NI, there exists a variable set $I \in \mathbb{I}_g$ such that the variable x in the gadget g is I-simulatable. It implies that the variable $x@\ell_g$ in the gadget f_{in} is $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g$-simulatable. The result immediately follows from the fact that $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \in \mathbb{I}_g[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g \subseteq \mathbb{I}_f$.
- Consider an inlined version $x@\ell_h$ of a variable x from the gadget h. We can get that the variable $x@\ell_h$ in the gadget f_{in} is $I[\vec{y}_1/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$-simulatable for $I \in \mathbb{I}_h$. Moreover, if the variable set I does not involve any shares of the encoding $\vec{b}_1$, then $x@\ell_h$ is $I[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$-simulatable. Otherwise the variable set I contains some shares of the encoding $\vec{b}_1$, since all the shares of the encoding $\vec{b}_1$ have the same uniform distribution, we can get that $x@\ell_h$ is $I[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$-simulatable as well. The result follows from the fact that $I[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h \in \mathbb{I}_h[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h \subseteq \mathbb{I}_f$.
- Consider an observable variable x defined in the gadget f. The variable x must be an actual parameter or a return value of one of the gadget call statements $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$.

  If x is an actual parameter of $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$, let c be the corresponding formal parameter of x. Then the variable c in the gadget g is I-simulatable for some $I \in \mathbb{I}_g$ or the variable c in the gadget h is I-simulatable for some $I \in \mathbb{I}_h$. Similarly, if the variable set I does not contain any shares of the encoding $\vec{b}_1$, we get that the variable x in the gadget f_{in} is I'-simulatable, where I' is $I[\vec{x}_1/\vec{a}_1, \cdots, \vec{x}_m/\vec{a}_m]@\ell_g$ or $I[\emptyset/\vec{b}_1, \vec{x}_2/\vec{b}_2, \cdots, \vec{x}_n/\vec{b}_n]@\ell_h$, since x and c always have the same value. Otherwise, the shares of the encoding $\vec{b}_1$ contained in the variable set I must have the same uniform distribution, we get that x in the gadget f_{in} is I'-simulatable as well.

If x is a return value of $\vec{y}_1 = g(\vec{x}_1, \cdots, \vec{x}_m)$ or $\vec{z} = h(\vec{y}_1, \vec{x}_2, \cdots, \vec{x}_n)$, let c be the corresponding return variable of x in the gadget g or h. We have proved that the variable $c@\ell_g$ (resp. $c@\ell_h$) in the gadget f_{in} is I-simulatable if c is a return variable of g (resp. h) for some $I \in \mathbb{I}_f$. Since x and c always have the same value, we get that the variable x in the gadget f_{in} is I-simulatable.

This completes the proof. □

Example 4.15. Let us consider the gadget call to `XORMULTI` in the running example. Since $\vec{a}$ and $\vec{b}$ are return encodings of two calls to the encoding gadget (i.e., `Encoding`), we have: $\lambda(\vec{a}) = \{\vec{a}\}$, $\lambda(\vec{b}) = \{\vec{b}\}$, and $\vec{a}$ and $\vec{b}$ are mutually independent. We show how to compute $\mathbb{I}_{\texttt{XORMULTI}}$ in this context.

(1) For $\vec{e}$ = `Refresh`$(\vec{a})$, as $\lambda(\vec{a}) = \{\vec{a}\}$ and $T_{\texttt{Refresh}} = \{\vec{a}, \vec{c}\}$, we get that $\Psi(\vec{a})$ holds, $\lambda(\vec{e}) = \{\vec{a}, \vec{e}\}$ and $\mathbb{I}_{\texttt{Refresh}}@8@4 = \mathbb{I}_{\texttt{Refresh}}[\emptyset/\vec{a}] = \emptyset$ according to Propositions 4.8, 4.10 and 4.14.
(2) For $\vec{c}$ = `XOR`$(\vec{b}, \vec{e})$, as $\lambda(\vec{b}) = \{\vec{b}\}$, $\lambda(\vec{e}) = \{\vec{a}, \vec{e}\}$ and $T_{\texttt{XOR}} = \{\vec{a}, \vec{b}\}$, we get: $\Psi(\vec{b}, \vec{e})$ holds, $\lambda(\vec{c}) = \{\vec{a}, \vec{b}, \vec{e}\}$ and $\mathbb{I}_{\texttt{XOR}}@9@4 = \mathbb{I}_{\texttt{XOR}}[\emptyset/\vec{a}, \emptyset/\vec{b}] = \emptyset$.
(3) For $\vec{d}$ = `UMA`$(\vec{e}, \vec{c})$, as $\lambda(\vec{e}) = \{\vec{a}, \vec{e}\}$ and $\lambda(\vec{c}) = \{\vec{a}, \vec{b}, \vec{e}\}$, we get: $\Psi(\vec{e}, \vec{c})$ holds, $\lambda(\vec{d}) = \vec{d}$ and $\mathbb{I}_{\texttt{UMA}}@10@4 = \mathbb{I}_{\texttt{UMA}}[\emptyset/\vec{a}, \emptyset/\vec{b}] = \emptyset$.

Finally, we obtain that $\mathbb{I}_{\texttt{XORMULTI}} = \emptyset$. Thus, `XORMULTI` is first-order probing secure. Compared with the result in Example 4.13 where $\lambda(\vec{a}) = \lambda(\vec{b}) = \emptyset$, we only deduced that $\lambda(\vec{e}) = \{\vec{e}\}$ and $\lambda(\vec{c}) = \emptyset$. Compared with the result in Example 3.5 where $\mathbb{I}_{\texttt{XORMULTI}} = \{\{\vec{a}[i]\}, \{\vec{b}[i], \vec{e}[i]\}, \{\vec{e}[i], \vec{c}[j]\} | 1 \leq i, j \leq 2\}$, $\mathbb{I}_{\texttt{XORMULTI}}$ turns to $\emptyset$ using the dominant variables of the actual parameters $\vec{a}$ and $\vec{b}$. □

5 ALGORITHMIC VERIFICATION

In this section, we first present a sound proof system for checking simulatability. Then we introduce algorithms to infer pre-conditions for simple gadgets and composite gadgets by utilizing the proof system and composition rules.

5.1 A Sound Proof System for Checking Simulatability

We first show how to use random dominant variables to simplify computations which is leveraged to derive valid judgements in our proof system. Given a sub-expression $e \in \mathsf{Sub}(x)$ and a random variable $r \in \mathsf{Var}(\mathcal{E}(x))$, let $x[r/e]$ denote the new variable x' such that $\mathcal{E}(x')$ is obtained by replacing e by r in $\mathcal{E}(x)$. (Note that $\mathsf{Sub}(x)$ denotes the set of the sub-expressions of $\mathcal{E}(x)$.) For each sub-expression $e \in \mathsf{Sub}(x)$, if $r \in \mathsf{DomR}(e)$ and r does not occur anywhere else in $\mathcal{E}(x)$, x can be simplified to $x' = x[r/e]$, denoted as $x \xrightarrow{(e,r)} x'$. This process can be repeated until there does not exist such random dominant variable r and sub-expression e. By Proposition 4.2, x is I-simulatable iff x' is I-simulatable.

$$\frac{x \in I \vee \mathsf{Var}(\mathcal{E}(x)) \subseteq I \cup X_r^{f_{\text{in}}}}{\vdash I \rightsquigarrow x}\ (\textsc{Supp}) \qquad \frac{x \xrightarrow{(e,r)} x' \quad \vdash I \rightsquigarrow x'}{\vdash I \rightsquigarrow x}\ (\textsc{Dom})$$

Fig. 6. Proof rules.

Given a gadget f, the judgment is in the form of

$$\vdash I \rightsquigarrow x$$

where $I \subseteq X_a^f \cup X^{f_{in}}$ and $x \in X^{f_{in}}$. The judgment $\vdash I \rightsquigarrow x$ is valid iff the variable x in the gadget f is I-simulatable.

Fig. 6 shows two proof rules for deriving valid judgment $\vdash I \rightsquigarrow x$. The first rule (i.e. Supp) exploits syntactic information and states that if the variable x is in I or the support variables of $\mathcal{E}(x)$ are either random variables or in I, then $\vdash I \rightsquigarrow x$ is valid. Another rule (i.e. Dom) makes use of semantic information. If a variable x can be simplified to x' using the random dominant variable r and $\vdash I \rightsquigarrow x'$ is valid, then $\vdash I \rightsquigarrow x$ is valid. This proof system will be used to prove $\mathbb{I}$-NI of simple gadgets in the next subsection.

Theorem 5.1. *If $\vdash I \rightsquigarrow x$ can be derived by the proof system, then the variable x in the gadget f is I-simulatable..*

Proof. It suffices to prove the soundness of the above two rules.

- Rule (Supp). Suppose the premise $x \in I \vee \mathsf{Var}(\mathcal{E}(x)) \subseteq I \cup X_r^{f_{\text{in}}}$ holds. If $x \in I$, since x is x-simulatable, by Proposition 2.3, we get that x is I-simulatable. If $\mathsf{Var}(\mathcal{E}(x)) \subseteq I \cup X_r^{f_{\text{in}}}$, then the values of the variables in $\mathsf{Var}(\mathcal{E}(x)) \setminus X_r^{f_{\text{in}}}$ can be directly obtained from the values of the variables in I. Since the values of the variables in $\mathsf{Var}(\mathcal{E}(x)) \cap X_r^{f_{\text{in}}}$ are uniformly sampled, knowing the values of I suffices to simulate the distribution of x, hence x is I-simulatable.
- Rule (Dom). Suppose $x \xrightarrow{(e,r)} x'$. By Proposition 4.2, we have: x is I-simulatable iff x' is I-simulatable. The result immediately follows.

We conclude the proof. □

Example 5.2. Consider the gadget UMA in the running example. From $\mathcal{E}(t_1) = \vec{a}[1] \odot \vec{b}[1]$, $\mathcal{E}(t_2) = \vec{a}[2] \odot \vec{b}[2]$, $\mathcal{E}(t_3) = \vec{a}[1] \odot \vec{b}[2]$, and $\mathcal{E}(t_4) = \vec{a}[2] \odot \vec{b}[1]$, by applying Rule (Supp), we can deduce that $\vdash \{\vec{a}[1], \vec{b}[1]\} \rightsquigarrow t_1$, $\vdash \{\vec{a}[2], \vec{b}[2]\} \rightsquigarrow t_2$, $\vdash \{\vec{a}[1], \vec{b}[2]\} \rightsquigarrow t_3$, and $\vdash \{\vec{a}[2], \vec{b}[1]\} \rightsquigarrow t_4$ are valid, thus t_1 is $\{\vec{a}[1], \vec{b}[1]\}$-simulatable, t_2 is $\{\vec{a}[2], \vec{b}[2]\}$-simulatable, t_3 is $\{\vec{a}[1], \vec{b}[2]\}$-simulatable, and t_4 is $\{\vec{a}[2], \vec{b}[1]\}$-simulatable.

Consider a variable x such that $\mathcal{E}(x) = (a \oplus r_1 \oplus r_2) \odot (b \oplus r_1)$, where r_1 and r_2 are random variables. Without applying Rule (Dom), we can only deduce that x is $\{a, b\}$-simulatable. By applying Rule (Dom), the sub-expression $(a \oplus r_1 \oplus r_2)$ can be replaced by the random variable r_2, leading to the simplified expression $\mathcal{E}(x') = r_2 \odot (b \oplus r_1)$, which further can be simplified to $\mathcal{E}(x'') = r_2 \odot r_1$. Obviously, $\vdash \emptyset \rightsquigarrow x''$ is valid, thus we deduce that all of x, x' and x'' are $\emptyset$-simulatable. □

5.2 Inferring Pre-conditions of Simple Gadgets

Algorithm 1 shows how to infer pre-conditions $\mathbb{I}$ for simple gadgets by leveraging the sound proof system to check simulatability. SGadget takes a simple gadget f as input, and outputs $(\mathbb{I}, T)$ such that f is $\mathbb{I}$-NI and T is the summary of the gadget f. We use the mapping `storedInfo` to store the checking results. Thus, if `storedInfo`(f) exists, it immediately returns the result stored in `storedInfo`. Otherwise, it computes $(\mathbb{I}, T)$ for the gadget f.

The pre-condition $\mathbb{I}$ is initialized (Line 6) by the following set

$$\{I \mid I = \mathsf{Var}(\mathcal{E}(x)) \subseteq X_a^f \wedge x \in X_a^f \cup X^f\}.$$

Intuitively, the computation $\mathcal{E}(x)$ of a variable x whose support variables are all the shares of input encodings cannot be simplified. Thus, $I = \mathsf{Var}(\mathcal{E}(x))$ should be added into the pre-condition $\mathbb{I}$. After initializing $\mathbb{I}$, to keep $\mathbb{I}$ as small as possible, if there exists a set $I \in \mathbb{I}$ such that I is the subset of $I' \in \mathbb{I}$, then I is removed from $\mathbb{I}$ (Line 8). For each variable $x \in X^f$, if there does not exist $I \in \mathbb{I}$ such that $\vdash I \rightsquigarrow x$ is valid via our proof rules, the set $\{x\}$ is directly added into the pre-condition $\mathbb{I}$ (Lines 9–11). Next, it checks whether f can transfer dominant variables from the input encoding $\vec{a}_i$ to the output encoding $\vec{o}$ for each $\vec{a}_i \in \{\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_m\}$. By Definition 4.4, if f

Algorithm 1 Checking simple gadget

1: **procedure** SGadget(f)
2: **if** storedInfo(f) exists **then**
3: **return** storedInfo(f)
4: Let $\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_m$ be the formal parameters of f
5: Let $\vec{o}$ be the output encoding of f
6: $\mathbb{I} = \{I \mid I = \mathsf{Var}(\mathcal{E}(x)) \subseteq X_a^f \wedge x \in (X_a^f \cup X^f)\}$
7: $T = \emptyset$
8: $\mathbb{I} = \mathbb{I} \setminus \{I \in \mathbb{I} \mid \exists I' \in \mathbb{I}. I \subset I'\}$
9: **for** each variable $x \in X^f$ **do**
10: **if** $\nexists I \in \mathbb{I}\ s.t.\ \vdash I \rightsquigarrow x$ **then**
11: $\mathbb{I} = \mathbb{I} \cup \{\{x\}\}$
12: **for** each $\vec{a}_i \in \{\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_m\}$ **do**
13: **if** f *can transfer dominant variables* from $\vec{a}_i$ to $\vec{o}$ **then**
14: $T = T \cup \{\vec{a}_i\}$
15: **if** f can *generates dominant variables* **then**
16: $T = T \cup \{\vec{o}\}$
17: storedInfo(f) = $(\mathbb{I}, T)$
18: **return** $(\mathbb{I}, T)$

can transfer dominant variables from $\vec{a}_i$ to the output encoding $\vec{o}$, the input encoding $\vec{a}_i$ is added into the summary T (Lines 12–14). Whether the input encoding $\vec{a}_i$ can transfer dominant variables to the output encoding $\vec{o}$ is checked according to Definition 4.1, aimed with the simplification $x \xrightarrow{(e,r)} x'$. By Definition 4.5, if f can generate dominant variables itself, the output encoding $\vec{o}$ is added into the summary T (Lines 15–16). Whether f can generate dominant variables itself is checked easily by checking if $\vdash \emptyset \rightsquigarrow \vec{o}[j']$ is valid for each share $\vec{o}[j']$ of the output encoding $\vec{o}$. Finally, the result $(\mathbb{I}, T)$ is stored in storedInfo and then returned. It is easy to see that f is $\mathbb{I}$-NI and T is the summary of f.

Example 5.3. Consider the simple gadget Refresh in the running example. Algorithm 1 first initializes the set $\mathbb{I}$ as $\{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}$ at Line 6. The set $\mathbb{I}$ cannot be simplified at Line 8. At Lines 9–11, the proof system will prove that $\vdash \emptyset \rightsquigarrow \vec{c}[1]$ and $\vdash \emptyset \rightsquigarrow \vec{c}[2]$ are valid, thus neither $\vec{c}[1]$ nor $\vec{c}[2]$ is added into the set $\mathbb{I}$. At Lines 12–14, the input encoding $\vec{a}$ is added into the summary T, because the share $\vec{c}[i]$ of the encoding $\vec{c}$ is dominated by only the share $\vec{a}[i]$ of the input encoding $\vec{a}$ for $i = 1, 2$. At Lines 15–16, the output encoding $\vec{c}$ is added into the summary T, because $\vdash \emptyset \rightsquigarrow \vec{c}[1]$ and $\vdash \emptyset \rightsquigarrow \vec{c}[2]$ are valid. Finally, Algorithm 1 returns the pair $(\{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}, \{\vec{a}, \vec{c}\})$.

Consider the gadget XOR in the running example. Algorithm 1 first initializes the set $\mathbb{I}$ as $\{\{\vec{a}[1]\}, \{\vec{a}[2]\}, \{\vec{b}[1]\}, \{\vec{b}[2]\}, \{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}$ at Line 6. At Line 8, the set $\mathbb{I}$ is simplified to $\{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}$. At Lines 9–11, the proof system will prove that $\vdash \{\vec{a}[1], \vec{b}[1]\} \rightsquigarrow \vec{c}[1]$ and $\vdash \{\vec{a}[2], \vec{b}[2]\} \rightsquigarrow \vec{c}[2]$ are valid, thus neither $\vec{c}[1]$ nor $\vec{c}[2]$ is added into the set $\mathbb{I}$. At Lines 12–14, the input encodings $\vec{a}$ and $\vec{b}$ are added into the summary T, because the share $\vec{c}[i]$ of the encoding $\vec{c}$ is dominated by only the share $\vec{a}[i]$ of $\vec{a}$ and the share $\vec{b}[i]$ of $\vec{b}$ for $i = 1, 2$. At Lines 15–16, the output encoding $\vec{c}$ cannot be added into the summary T, because neither $\vdash \emptyset \rightsquigarrow \vec{c}[1]$ nor $\vdash \emptyset \rightsquigarrow \vec{c}[2]$ is valid. Finally, Algorithm 1 returns the pair $(\{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}, \{\vec{a}, \vec{b}\})$.

Similarly, Algorithm 1 returns $(\{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}, \{\vec{a}[1], \vec{b}[2]\}, \{\vec{a}[2], \vec{b}[1]\}\}, \{(t_6, t_8)\})$ for the gadget UMA.

Algorithm 2 Checking composite gadget

1: **procedure** GADGET(f, λ)
2: **if** f is a simple gadget **then** $(\mathbb{I}, T)$ =SGADGET(f)
3: **else** $(\mathbb{I}, T)$ =CGADGET(f, λ)
4: **return** $(\mathbb{I}, T)$
5: **procedure** CGADGET(f, λ)
6: **if** storedInfo(f, λ) exists **then**
7: **return** storedInfo(f, λ)
8: $\mathbb{I} = \emptyset$
9: $\lambda_1 = \lambda$
10: Let $\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_n$ be the formal parameters of f
11: Let $\vec{o}_1$ be the output encoding of f
12: **for** each gadget call $\vec{y} = g(\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m)$ at ℓ_g from the first to the last **do**
13: Let $\vec{b}_1, \cdots, \vec{b}_m$ be the formal parameters of g
14: Let $\vec{o}_2$ be the output encoding of g
15: **for** $i \in \{1, \cdots, m\}$ **do**
16: **if** $\exists \vec{x} \in \lambda(\vec{x}_i) \cap \lambda(\vec{x})$ **then** $\lambda'(\vec{b}_i) = \{\vec{b}_i\}$
17: **else** $\lambda'(\vec{b}_i) = \emptyset$
18: $(\mathbb{I}', T') = $ GADGET(g, λ')
19: **if** $\vec{x}_1, \cdots, \vec{x}_m$ are distinct **then**
20: **if** $\Psi(\vec{x}_1, ..., \vec{x}_m)$ or $\vec{x}_1 \in \lambda(\vec{x}_1), ..., \vec{x}_m \in \lambda(\vec{x}_m)$ **then**
21: **for** $i \in \{1, ..., m\}$ **do**
22: **if** $\vec{b}_i \in T'$ **then** $\lambda(\vec{y}) = \lambda(\vec{y}) \cup \lambda(\vec{x}_i)$
23: **if** $\vec{o}_2 \in T'$ **then**
24: $\lambda(\vec{y}) = \lambda(\vec{y}) \cup \{\vec{y}\}$
25: **for** $i \in \{1, \cdots, m\}$ **do**
26: $\vec{x}'_i = (\exists \vec{x} \in \lambda(\vec{x}_i) \cap \lambda(\vec{x})\ ?\ \emptyset\ :\ \vec{x}_i)$
27: $\mathbb{I} = \mathbb{I} \cup \mathbb{I}'[\vec{x}'_1/\vec{b}_1, \cdots, \vec{x}'_m/\vec{b}_m]@\ell_g$
28: $T = \lambda(\vec{o}_1)$
29: **if** $T \setminus \{\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_n, \vec{o}_1\} \neq \emptyset$ **then**
30: $T = (T \cap \{\vec{a}_1, \vec{a}_2, \cdots, \vec{a}_n, \vec{o}_1\}) \cup \{\vec{o}_1\}$
31: storedInfo(f, λ_1) = $(\mathbb{I}, T)$
32: **return** $(\mathbb{I}, T)$

These results are consistent with the ones given in Example 3.3, namely, Refresh is $\mathbb{I}_{\texttt{Refresh}}$-NI, XOR is $\mathbb{I}_{\texttt{XOR}}$-NI and UMA is $\mathbb{I}_{\texttt{UMA}}$-NI, where

- $\mathbb{I}_{\texttt{Refresh}} = \{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}$;
- $\mathbb{I}_{\texttt{XOR}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}\}$;
- $\mathbb{I}_{\texttt{UMA}} = \{\{\vec{a}[1], \vec{b}[1]\}, \{\vec{a}[2], \vec{b}[2]\}, \{\vec{a}[1], \vec{b}[2]\}, \{\vec{a}[2], \vec{b}[1]\}\}$. □

5.3 Inferring Pre-conditions for Composite Gadgets

We propose two procedures GADGET and CGADGET in Algorithm 2, where the procedure GADGET takes a gadget f and a function λ as input and outputs the result $(\mathbb{I}, T)$ such that f is $\mathbb{I}$-NI and T is the summary of the gadget f. If f is a simple gadget, then the procedure GADGET calls the procedure SGADGET at Line 2 (cf. Algorithm 1) and returns the result immediately. Otherwise,

it calls the procedure CGadget at Line 3 which takes a composite gadget f and λ as input and outputs $(\mathbb{I}, T)$ such that the gadget f is $\mathbb{I}$-NI and T is the summary of the gadget f.

Recall that λ is a function used for computing summaries of gadgets (cf. Section 4.2.2). For each encoding $\vec{a}$, $\lambda(\vec{a})$ contains all the dominant encodings of $\vec{a}$, and moreover $\vec{a} \in \lambda(\vec{a})$ only if $\vec{a}$ is the return of a gadget call that can generate dominant variables. However, λ is unknown in advance, thus we will compute the function λ during analysis for each composite gadget. We also use the mapping `storedInfo` to store the checking result, namely $\texttt{storedInfo}(f, \lambda)$ records the result of $\textsc{CGadget}(f, \lambda)$, where λ initially contains only the dominant encodings of the formal parameters of the gadget f and thus can be seen as the calling context of the gadget f. We note that the same gadget may be called multiple times with different calling contexts, thus λ is involved in `storedInfo`.

The procedure CGadget first checks if $\texttt{storedInfo}(f, \lambda)$ exists or not, it returns $\texttt{storedInfo}(f, \lambda)$ if it exists (Line 7). Otherwise, it reasons about the gadget f with the function λ. First, the pre-condition $\mathbb{I}$ is set to $\emptyset$ (Line 8) and λ_1 is set to λ as a backup of calling context λ (Line 9). For each gadget call $\vec{y} = g(\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m)$ at call-site ℓ_g, it computes the calling context λ' based on $\lambda(\vec{x}_i)$ (Lines 15–17), namely, for each formal parameter $\vec{b}_i$, $\lambda'(\vec{b}_i)$ is $\{\vec{b}_i\}$ if $\exists \vec{x} \in \lambda(\vec{x}_i) \cap \lambda(\vec{x})$, otherwise $\emptyset$, where $\vec{x} \in \lambda(\vec{x}_i)$ indicates that the actual parameter $\vec{x}_i$ is dominated by the encoding $\vec{x}$ and $\vec{x} \in \lambda(\vec{x})$ indicates that $\vec{x}$ is the return of a gadget call to a gadget that generates dominant variables.

After constructing λ', the result $(\mathbb{I}', T')$ of the gadget g under λ' is obtained by invoking $\textsc{Gadget}(g, \lambda')$ (Line 18). Then, by Proposition 4.10, if $\vec{x}_1, \cdots, \vec{x}_m$ are mutually independent and (either $\Psi(\vec{x}_1, \cdots, \vec{x}_m)$ holds or $\vec{x}_1 \in \lambda(\vec{x}_1), \cdots, \vec{x}_m \in \lambda(\vec{x}_m)$), for each formal parameter $\vec{b}_i \in T'$, the set $\lambda(\vec{x}_i)$ is merged into $\lambda(\vec{y})$ (Line 22). Next, by Proposition 4.8, if g can generate dominant variables itself, i.e., $\vec{o}_2 \in T'$, $\vec{y}$ is added into $\lambda(\vec{y})$ too (Line 24). After that, the pre-condition $\mathbb{I}$ is computed following Proposition 4.14 (Line 27).

After computing $\mathbb{I}$, it continues to compute the summary T of the gadget f which is initialized as $\lambda(\vec{o}_1)$. To ensure that $T \subseteq X^f_{en} \cup \{\vec{o}_1\}$, it removes all the internal encodings of f from the summary T and uses $\vec{o}_1$ to indicate that f can generate dominant variables itself (cf. Proposition 4.7). Finally, the result $(\mathbb{I}, T)$ is stored in `storedInfo` and returned. Note that after computing $\mathbb{I}$, λ has been updated. Thus, we use its backup λ_1 to store the result (Line 31).

Theorem 5.4. *Given a program P with the* `main` *gadget*

$$\texttt{main}(a_1, \cdots, a_m)\{\texttt{enstmt}^+\ \texttt{gstmt}^+\ \texttt{return}\ \vec{b};\},$$

let g be the gadget $g(\vec{a}_1, \cdots, \vec{a}_m)\{\texttt{gstmt}^+\ \texttt{return}\ \vec{b};\}$ with the same gadget calls $\texttt{gstmt}^+$ *as* `main`*, where $\vec{a}_1, \cdots, \vec{a}_m$ are the encodings of $a_1, \cdots, a_m$ via calling an encoding gadget (e.g.,* `Encoding`*). Let λ be a function such that $\lambda(\vec{a}_i) = \{\vec{a}_i\}$ for $\vec{a}_i \in \{\vec{a}_1, \cdots, \vec{a}_m\}$. If $(\mathbb{I}, T) = \textsc{Gadget}(g, \lambda)$ and each variable set $I \in \mathbb{I}$ is X_p-simulatable, then P is first-order probing secure.*

Proof. Since $(\mathbb{I}, T) = \textsc{Gadget}(g, \lambda)$, we get that g is $\mathbb{I}$-NI. The result immediately follows from Proposition 3.2 and the fact that $\vec{a}_1, \cdots, \vec{a}_m$ are the encodings of $a_1, \cdots, a_m$ via calling an encoding gadget (e.g., `Encoding`). □

Example 5.5. Consider the composite gadget `XORMULTI` in the running example which is called in the `main` gadget. Procedure CGadget in Algorithm 2 is invoked for the gadget `XORMULTI` with $\lambda(\vec{a}) = \{\vec{a}\}$ and $\lambda(\vec{b}) = \{\vec{b}\}$. During the for-loop at Lines 12–27, the gadget call $\vec{e} = \texttt{Refresh}(\vec{a})$ is firstly processed as follows:

- Since $\lambda(\vec{a}) = \{\vec{a}\}$, we have: $\lambda'(\vec{a}) = \{\vec{a}\}$ at Lines 15–17.
- The pair $(\{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}, \{\vec{a}, \vec{c}\})$ is returned at Line 18 (cf. Example 5.3).

- Since $\lambda(\vec{a}) = \{\vec{a}\}$ and $\vec{a} \in T' = \{\vec{a}, \vec{c}\}$, $\lambda(\vec{e})$ is set to $\lambda(\vec{e}) \cup \lambda(\vec{a})$, that is $\{\vec{a}\}$, at Line 22.
- Since $\vec{c} \in T' = \{\vec{a}, \vec{c}\}$, $\{\vec{e}\}$ is added into $\lambda(\vec{e})$, leading to $\lambda(\vec{e}) = \{\vec{a}, \vec{e}\}$ at Line 24.
- Since $\lambda(\vec{a}) = \{\vec{a}\}$, $\vec{x}'_i$ is set to $\emptyset$ at Line 26.
- Finally, $\mathbb{I}$ is set to $\mathbb{I} \cup \{\{\vec{a}[1]\}, \{\vec{a}[2]\}\}[\emptyset/\vec{a}]@\ell_g$ that is $\emptyset$ at Line 27

Later, the gadget calls $\vec{e} = \texttt{Refresh}(\vec{a})$, $\vec{c} = \texttt{XOR}(\vec{b}, \vec{e})$ and $\vec{d} = \texttt{UMA}(\vec{e}, \vec{c})$ are iteratively processed similar to the gadget call $\vec{e} = \texttt{Refresh}(\vec{a})$. At the exit of the for-loop at Lines 12–27, $\mathbb{I}$ is still $\emptyset$ and $\lambda(\vec{d}) = \{\vec{d}\}$. Finally, procedure CGadget returns the pair $(\emptyset, \{d\})$.

This result is consistent with the one given in Example 4.15, namely, XORMULTI is $\emptyset$-NI. □

We remark that pre-conditions of both simple and composite gadgets can always be successfully and automatically computed without inlining gadget calls and user interactions. However, technically speaking, in the worst case, the pre-condition of a simple gadget g may contain all the input parameters and internal variables. For instance, if the computation $\mathcal{E}(x)$ of each internal variable x uses some random variables but is not dominated by any random variables, then $\mathbb{I} = \{\{a\} \mid a \in X^g_a\}$ at Line 6 of Algorithm 1 and further the sets $\{x\}$ for all $x \in X^g$ will be added into the set $\mathbb{I}$ at Lines 9–11 of Algorithm 1. Hence, the set $\mathbb{I}'$ obtained at Line 18 of Algorithm 2 will be $\{\{x\} \mid x \in X^g_a \cup X^g\}$. If the worst case occurs for all the simple gadgets, the pre-condition $\mathbb{I}$ of a composite gadget f will contain all the input parameters and internal variables of its inlined version f_{in}, which is the same as that the pre-condition of f is computed on the inlined version f_{in}. We should emphasize that the worst-case never occurs in our experiments, because the computations $\mathcal{E}(x)$ of internal variables typically either depend only on input parameters (in this case $\mathsf{Var}(\mathcal{E}(x)) \subseteq X^f_a$ is added into the set $\mathbb{I}$ at Line 6 of Algorithm 1) or are perfectly masked by XORing random variables (in this case $\vdash \emptyset \rightsquigarrow x$ is valid at Lines 9–11 of Algorithm 1 and $\{x\}$ will not be added into $\mathbb{I}$).

6 EXPERIMENTS

We have implemented our approach in a tool named MaskCV. Given a masked program and its security type annotation of input parameters, MaskCV first preprocesses the program by unfolding bounded for-loops and transforming into an intermediate representation in SSA form, then infers pre-conditions and verifies first-order probing security, all of which are done automatically without any user interactions. In this section, we thoroughly evaluate MaskCV focusing on the following research questions:

RQ1: How efficient is our approach compared to the state-of-the-art approaches?
RQ2: How effective are the dominated variables?

We mainly use the publicly avaiable benchmarks from QMVerif [53], i.e., 10 versions of masked implementations of arithmetic AES Sbox and 10 corresponding masked implementations of full AES. We also implement two versions of bitsliced AES Sbox. The bitsliced AES Sbox was originally proposed in [57] which contains multiple gadget calls to the `Refresh` gadget. Later Belaid et al. [18] proved that the bitsliced AES Sbox is still secure if all the calls to the `Refresh` gadget are removed, leading to an efficient and tight bitsliced AES Sbox in terms of gadget calls to the `Refresh` gadget. Moreover, we notice that the efficiency in terms of the randomness of the tight bitsliced AES Sbox can be improved further by replacing the calls to the `SecMult` gadget by more efficient secure gadgets, i.e., `Para` [11] and `Comp` [14]. Thus, we construct two implementations of bitsliced AES Sbox `bitslicedSbox1` and `bitslicedSbox2` by replacing the calls to `SecMult` by `Comp` and `Para`, respectively. We remark that the first-order security of all those benchmarks cannot be verified by any of the existing compositional verifiers though some of them support the verification of higher-order security, except for QMVerif. Details of the benchmarks are shown in Table 2, including the number of gadgets, the number of gadget calls and the size of annotations used for QMVerif.

Table 2. Basic Information of benchmarks

Name	♯Gadgets	♯Gadget calls	♯Annotations for QMVerif
Sbox1	10	14	18
Sbox2	10	14	18
Sbox3	11	12	22
Sbox4	11	12	22
Sbox5	10	14	30
Sbox6	10	14	30
Sbox7	10	14	30
Sbox8	10	14	30
Sbox9	11	12	34
Sbox10	11	12	34
AES1	20	3,433	178
AES2	20	3,433	146
AES3	21	3,033	152
AES4	21	3,033	150
AES5	20	3,433	206
AES6	20	3,433	206
AES7	20	3,433	206
AES8	20	3,433	206
AES9	21	3,033	228
AES10	21	3,033	228
bitslicedSbox1	5	125	78
bitslicedSbox2	5	125	78

All experiments were conducted on a server with Ubuntu 16.04, Intel(R) Xeon(R) CPU E5-2690 v4@2.60GHz and 256GB RAM. The reported verification time includes all the computation times for preprocessing, determining dominated variables, computing pre-conditions and verifying first-order probing security based on pre-conditions.

RQ1. We compare with the state-of-the-art compositional verifier QMVerif and the state-of-the-art non-compositional ones `LeakageVerif` [77] and `SILVER` [69], where `LeakageVerif` uses symbolic analysis and `SILVER` is based on BDD analysis. Note that `SILVER` only supports Boolean programs, thus, we evaluate it only on Boolean programs (i.e., `bitslicedSbox1` and `bitslicedSbox2`) and mark N/A on arithmetic programs. All the tools QMVerif, `LeakageVerif`, `SILVER` and MaskCV need loop unfolding and all bounded loops are automatically unfolded. QMVerif, `SILVER` and MaskCV require the SSA form and support automatically SSA transformation, while LeakageVerif does not need the SSA form.

The results are shown in Table 3, in which T.O. stands for "time-out" (6 hours) and O.O.M. stands for "out of memory". Column 1 shows the benchmark name; Column 2 shows the verification result; Column 3 and Column 4 (resp. Column 5 and Column 6) show the verification time in seconds (s) and number of variables checked by QMVerif without pre-conditions (resp. with all the pre-conditions); Columns 7 and 8 show the verification time and the number of variables checked by `LeakageVerif`; Column 9 shows the verification time of `SILVER`. Column 10 and 11 show the verification time and number of variables checked by MaskCV with dominated variables (that are the variables checked at Lines 9–11 in Algorithm 1 and variables of the pre-condition I used for checking first-order probing security of the program). Recall that the verification time reported in Table 3 includes all the computation times for preprocessing, determining dominated variables, computing pre-conditions and verifying first-order probing security based on pre-conditions. Since MaskCV is able to solve each benchmark in no more than 0.04 seconds, we did not report the individual computational time for each step.

Table 3. Results of comparison with Existing Approaches, where Time(s) denotes the overall verification time in seconds (s), T.O. denotes time-out (6 hours), O.O.M. denotes out of memory, and ♯checked denotes the number of checked variables

Name	Result	QMVerif [53]				LeakageVerif [77]		SILVER [69]	MaskCV	
		Without Pre-conditions		With Pre-conditions						
		Time(s)	♯checked	Time(s)	♯checked	Time(s)	♯checked	Time(s)	Time(s)	♯checked
Sbox1	✓	0.01	46	0.01	19	0.04	51	N/A	0.01	22
Sbox2	✓	0.01	50	0.01	19	0.07	43	N/A	0.01	23
Sbox3	✓	0.01	70	0.01	60	0.08	65	N/A	0.01	68
Sbox4	✓	0.01	68	0.01	60	0.09	69	N/A	0.01	67
Sbox5	✓	0.01	110	0.01	42	0.15	115	N/A	0.01	45
Sbox6	✓	0.01	122	0.01	42	0.10	103	N/A	0.01	48
Sbox7	✓	0.01	118	0.01	50	0.17	167	N/A	0.01	48
Sbox8	✓	0.01	130	0.01	50	0.19	173	N/A	0.01	51
Sbox9	✓	0.01	178	0.01	150	0.16	162	N/A	0.01	174
Sbox10	✓	0.01	172	0.01	150	0.16	162	N/A	0.01	171
AES1	✓	315	11,142	0.10	2,182	559	10,706	N/A	0.02	29
AES2	✓	197	11,942	0.10	2,183	561	10,706	N/A	0.02	30
AES3	✓	559	15,942	0.10	2,393	1,836	14,706	N/A	0.02	75
AES4	✓	560	15,542	0.10	2,392	2,144	14,706	N/A	0.02	74
AES5	✓	2,670	24,724	0.40	4,214	T.O.	N/A	N/A	0.03	55
AES6	✓	3,505	27,124	0.40	4,214	T.O.	N/A	N/A	0.03	58
AES7	✓	2,933	26,324	0.50	4,430	T.O.	N/A	N/A	0.03	58
AES8	✓	3,130	28,724	0.50	4,430	T.O.	N/A	N/A	0.03	61
AES9	✓	2,929	38,324	0.20	3,786	O.O.M.	N/A	N/A	0.03	184
AES10	✓	3,064	37,124	0.20	3,786	O.O.M.	N/A	N/A	0.04	181
bitslicedSbox1	✓	0.02	961	0.02	451	9.30	873	O.O.M.	0.01	27
bitslicedSbox2	✓	0.02	1,057	0.02	453	8.47	937	T.O.	0.01	30

We can observe that both MaskCV and QMVerif are able to prove all those benchmarks. But, MaskCV is significantly more efficient than QMVerif without pre-conditions on large benchmarks that have a large number of gadget calls (e.g., AES1–AES10) while is comparable or slightly better when all the pre-conditions are provided to QMVerif. This indicates the effectiveness of our pre-condition inference approach. By comparing the number of checked variables using QMVerif with and without user-provided pre-conditions, the number of checked variables (thus the verification time) is significantly reduced with user-provided pre-conditions. It is because a gadget call needs not be inlined if the user-provided pre-condition is satisfied by the actual parameters of the gadget call (but has to be inlined if the user-provided pre-condition is not satisfied by the actual parameters of the gadget call). Our tool MaskCV automatically infers pre-conditions of all the gadgets, thus needs not check whether the inferred pre-conditions are satisfied by the actual parameters of the gadget calls, which further reduces the number of checked variables. Thus, MaskCV is comparable or slightly better than QMVerif with all the pre-conditions, and significantly more efficient than QMVerif without the pre-conditions on large benchmarks. We should emphasize that MaskCV is much easier to use as it is fully automatic whereas QMVerif needs user-provided pre-conditions to be efficient.

Compared with the state-of-the-art non-compositional verifiers `LeakageVerif` and `SILVER`, MaskCV is more efficient, in particular, on large benchmarks (e.g., AES1–AES10). The reason is that `LeakageVerif` checks all the observable variables after inlining gadget calls and the number of observable variables increases quickly (exponentially in the worst case) after inlining gadget calls, while MaskCV can directly verify composite gadgets without inlining them. `SILVER` fails to verify both `bitslicedSbox1` and `bitslicedSbox2` because it fails to construct the BDD models due to the large number of observable variables and the large size of computations.

Table 4. Results of MaskCV with and without leveraging dominated variables, where ♯Gadgets denotes the number of the gadgets, ♯G-Gadgets denotes the number of the gadgets that can generate dominant variables, ♯T-Gadgets denotes the number of the gadgets that can transfer dominant variables, ♯G&T-Gadgets denotes the number of the gadgets that can both generate and transfer dominant variables, and $|\mathbb{I}|$ denotes the size of the pre-condition of the program on which the first-order probing security is verified.

Name	♯Gadgets	♯G-Gadgets	♯T-Gadgets	♯G&T-Gadgets	With Dom			Without Dom		
					Time(s)	$\lvert\mathbb{I}\rvert$	♯checked	Time(s)	$\lvert\mathbb{I}\rvert$	♯checked
Sbox1	10	5	7	3	0.01	0	22	0.01	28	66
Sbox2	10	5	7	3	0.01	0	23	0.01	28	67
Sbox3	11	6	6	2	0.01	0	68	0.01	20	96
Sbox4	11	6	6	2	0.01	0	67	0.01	20	95
Sbox5	10	5	7	3	0.01	0	45	0.01	54	135
Sbox6	10	5	7	3	0.01	0	48	0.01	54	138
Sbox7	10	5	7	3	0.01	0	48	0.01	54	138
Sbox8	10	5	7	3	0.01	0	51	0.01	54	141
Sbox9	11	6	6	2	0.01	0	174	0.01	36	228
Sbox10	11	6	6	2	0.01	0	171	0.01	36	225
AES1	20	8	13	5	0.02	0	29	60	7,836	12,673
AES2	20	8	13	5	0.02	0	30	48	7,836	12,674
AES3	21	9	12	4	0.02	0	75	51	6,236	9,519
AES4	21	9	12	4	0.02	0	74	55	6,236	9,518
AES5	20	8	13	5	0.03	0	55	262	14,154	23,821
AES6	20	8	13	5	0.03	0	58	262	14,154	23,824
AES7	20	8	13	5	0.03	0	58	266	14,154	23,824
AES8	20	8	13	5	0.03	0	61	284	14,154	23,827
AES9	21	9	12	4	0.03	0	184	211	10,554	16,750
AES10	21	9	12	4	0.04	0	181	236	10,554	16,747
bitslicedSbox1	5	3	2	1	0.01	0	27	0.38	537	1,101
bitslicedSbox2	5	3	2	1	0.01	0	30	0.37	537	1,104

Answer to RQ1: Our method is significantly more efficient than the state-of-the-art non-compositional approaches and state-of-the-art compositional approach without user-defined pre-conditions while achieves competitive efficiency compared with state-of-the-art compositional approach with user-defined pre-conditions.

RQ2. Table 4 presents the results of MaskCV with and without leveraging dominated variables. Column 1 shows the benchmark name; Columns 2, 3, 4 and 5 show the number of gadgets (♯Gadgets), the number of gadgets that can generate dominant variables (♯G-Gadgets), the number of gadgets that can transfer dominant variables (♯T-Gadgets), and the number of gadgets that can both generate and transfer dominant variables (♯G&T-Gadgets); and Columns 6, 7 and 8 (resp. Columns 9, 10 and 11) show the verification time, the size of the automatically inferred pre-condition $\mathbb{I}$ of the program on which first-order security is checked, and the number of variables checked by leveraging dominated variables (resp. without leveraging dominated variables).

We can observe that dominated variables can significantly reduce the size of the automatically inferred pre-condition $\mathbb{I}$ of the program, which in turn significantly reduces the number of variables that need to be checked when verifying first-order probing security of the program using the pre-condition $\mathbb{I}$. Indeed, all the final pre-conditions $\mathbb{I}$ of the programs by leveraging dominated variables become the empty set, because a large number of gadgets can generate dominant variables and/or transfer dominant variables variables from input parameters to the output encoding. It means that only the variables of simple gadgets are checked at Lines 9–11 in Algorithm 1 while the verification of the first-order probing security of the program based on the pre-condition $\mathbb{I}$ is avoided because of $\mathbb{I} = \emptyset$. Thus, the verification time is significantly reduced by leveraging dominated variables.

```
1  power254(x⃗){
2    z⃗ = power2(x⃗);
3    z⃗ = Refresh(z⃗);
4    y⃗ = SecMult(z⃗, x⃗);
5    w⃗ = power4(y⃗);
6    w⃗ = Refresh(w⃗);
7    y⃗ = SecMult(y⃗, w⃗);
8    y⃗ = power16(y⃗);
9    y⃗ = SecMult(y⃗, w⃗);
10   y⃗ = SecMult(y⃗, z⃗);
11   return y⃗;
12 }
```

Fig. 7. A composite gadget defined in Sbox1

Case study. We exemplify the advantage of dominated variables using the composite gadget power254 shown in Fig. 7, which is taken from the benchmark Sbox1. The simple gadgets power2, power4 and power16 compute an encoding $\vec{y}$ for a given encoding $\vec{x}$ such that $\bigoplus \vec{y} = (\bigoplus \vec{x})^2$, $\bigoplus \vec{y} = (\bigoplus \vec{x})^4$, and $\bigoplus \vec{y} = (\bigoplus \vec{x})^{16}$, respectively; and the simple gadget SecMult computes an encoding $\vec{z}$ for two given encodings $\vec{x}$ and $\vec{y}$ such that $\bigoplus \vec{z} = (\bigoplus \vec{x}) \odot (\bigoplus \vec{y})$.

The sizes of the inferred pre-conditions of the gadgets Refresh, power2, power4, power16 and SecMult by Algorithm 1 are 2, 2, 2, 2 and 4, respectively. Without leveraging dominated variables, the size of the inferred pre-condition $\mathbb{I}$ of the composite gadget power254 by Algorithm 2 is 26, as all the pre-conditions of the gadgets in the nine gadget calls are added into $\mathbb{I}$.

Since the gadgets Refresh and SecMult are capable of generating dominant variables, and all the gadgets Refresh, power2, power4 and power16 can also transfer dominant variables, by leveraging dominated variables, only the pre-conditions of the gadgets power2 and Refresh and part of the pre-condition of the gadget SecMult for the first three gadget calls are added into the pre-condition $\mathbb{I}$ of the gadget power254, while the pre-conditions of the gadgets power4, Refresh, power16, and SecMult in the remaining six gadget calls are not added, because $\vec{x}'_i$ is $\emptyset$ at Line 26 of Algorithm 2. Thus, the size of the inferred pre-condition $\mathbb{I}$ of power254 by Algorithm 2 is only 6.

Furthermore, when the gadget power254 is invoked with an actual parameter that is an dominated encoding, the calling context λ of the gadget power254 will be $\lambda(\vec{x}) = \{\vec{x}\}$. Using this additional information, the pre-conditions of the gadgets power2, Refresh and SecMult for the first three gadget calls are not added into the pre-condition $\mathbb{I}$ of power254 either. Thus, the pre-condition $\mathbb{I}$ of the gadget power254 inferred by Algorithm 2 will be $\emptyset$.

Answer to RQ2: Dominated variables effectively reduces the size of pre-conditions, hence the verification time.

Threats to validity. Our work focuses on cryptographic programs which, unlike general-purpose software, are structurally simple. The design of the specific program syntax tailored to the masked implementations of cryptographic algorithms has been thoroughly discussed in Section 2.1. In particular, we do not consider conditionals (e.g., if-then-else) which often induce timing side-channel leaks [71, 96], particularly under speculative execution [32, 70]. To avoid such leaks, programs often follow the constant-time principle to avoid conditionals. Therefore, it is not a real limitation and actually has been widely adopted in the literature [8, 9, 12, 17, 18, 24, 26, 30, 31, 38, 47, 69, 98]. For convenience, our tool supports bounded for-loops which are automatically and fully unfolded before verification, so we only present the core language without loops.

We did not compare with the other state-of-the-art compositional approaches such as [9, 10, 18, 31], because all the existing compositional approaches (except for QMVerif) cannot verify the first-order security of the publicly available benchmarks considered in this work. We did not consider other benchmarks that can be handled by existing compositional approaches, as the main purpose of the current work is to handle programs that *cannot* be proved by the existing compositional approaches. Essentially, we gain generality of implementations but may sacrifice verification performance in some cases. For instance, the implementations of cryptographic algorithms only use Boolean operations and moreover, each encoding is re-masked by using new random variables (e.g., invoking a `refresh` gadget) before it is used in the second place. Such Boolean programs satisfy these implicitly imposed pre-conditions in the *stronger* security notions [9, 10, 12, 16–19, 25, 26, 69], thus can be quickly proved using the existing compositional approaches, e.g., maskVerif [10], at the cost of the efficiency of the masked implementations.

A limitation of our work is we only deal with first-order secure programs which may be still vulnerable against higher-order power side-channel attacks. However, we note that (1) higher-order attacks are much more difficult to launch successfully, and (2) higher-order secure programs incur excessive overhead. As a result, first-order security is more suitable especially when the efficiency is the key. Our work essentially targets at these types of applications run in, e.g., resource-limited devices [20].

7 RELATED WORK

Along with the design of efficient masked implementations, various verification approaches have been proposed. Non-compositional approaches must inline gadget calls which would increase the size of the program and thus the verification cost. Some representative works are symbolic analysis [21, 38, 76, 77, 81, 91, 92], the SAT/SMT-based approaches SAT/SMT-based analysis [23, 47–50], BDD analysis [69] and hybrid approaches combine the symbolic analysis and SAT/SMT-based approach together [52, 54, 55, 98]. Symbolic analysis based approaches are more efficient than SAT/SMT-based approaches, however only provide soundness, while SAT/SMT-based approaches provide both soundness and completeness in theory but limit in scalability. Hybrid approaches are aimed to bring the best of both worlds, where an SAT/SMT-based approach is applied only when the symbolic analysis fails.

Compositional approaches directly check composite gadgets, date back to [9] in which stronger security notions and a sound proof system are proposed. Basically, when a return encoding is used as actual parameters multiple times, their approach directly sums up all the possible number of observable variables, leading to an inaccurate conclusion. We detail in Appendix A why their proof system fails to prove the running example in Fig. 5. Along this direction, various approaches have been proposed to check simple gadgets [10, 12, 26, 38, 69] w.r.t. the security notions of [9]. `SILVER` [69] supports probing security, but fails to prove first-order probing security in our experiments. Belaïd et al. [17–19] presented alternative composition approaches of [9] via matrix analysis, however, it assumes that all the gadgets are ISW gadgets [12], while some efficient gadgets that are non-ISW, e.g., the first-order masked AND gadget [20]. Cassiers et al. [30, 31] proposed an alternative stronger security notion PINI such that gadgets composed by PINI gadgets are still PINI. While all these compositional approaches support higher-order security, only bitwise logical operations are focused except for [38] and the pre-conditions in those security notions are fixed, which cannot be fulfilled by some efficient gadgets, where [38] is semi-automatic. Our approach is wider (e.g., arithmetic operations and non-ISW gadgets) for the first-order security of composite gadgets. The approach proposed by Gao et al. [53] is applicable to general first-order masked implementations, but it requires user-defined pre-conditions to be efficient. Our work bridges

the gap that there lacks an efficient compositional approach that is applicable to general gadgets without user-defined pre-conditions.

While mitigating techniques have been studied to eliminate power side-channel leaks [2, 9, 13, 17, 25, 46, 92], they either do not use formal verification to provide guarantees [2, 13] or rely on formal verification [9, 17, 25, 46, 92]. We focus on verification, but could also be extended to eliminate power side-channel leaks, similar to [9, 17].

We note that there are other types of side-channel attacks such as timing side-channel attacks which have been widely studied, including detection [7, 29, 79, 82, 83], verification [4, 6, 22, 33, 42, 44, 45, 61] and mitigation [75, 95, 96]. The more recent work focuses on timing side-channel leaks introduced by micro-architectural features [34, 43, 59, 60, 70, 73] and (JIT) compilation [27, 28, 42, 43, 85], and network-based side channel leaks [65, 66, 88] which contains a large number of observable aspects (such as the time, size and direction of each packet). As different types of side-channel attacks have their own characteristics, they are orthogonal to our work. There are other power leakage models and security notions [10, 16, 23, 51, 78, 92] which are however out of the scope of this paper. Finally, we remark that this work considers the first-order probing security of programs written in our domain-specific language while secure programs may become insecure after compilation [92], the same as in timing side-channel security setting.

8 CONCLUSION

We proposed a novel approach and algorithms to infer pre-conditions for compositional verification of generic and efficient masked implementations. We implemented our approach in a tool and conducted extensive experiments on publicly available benchmarks. It significantly outperforms the state-of-the-arts on masked implementations of full AES when no pre-conditions are provided by users.

ACKNOWLEDGEMENT

This work is supported by the National Natural Science Foundation of China (NSFC) under Grants No. 62072309 and No. 61872340, the Strategic Priority Research Program of the Chinese Academy of Sciences Grant No. XDA0320101, an overseas grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03, KFKT2023A04), and Birkbeck BEI School Project (EFFECT).

REFERENCES

[1] 2022. *MASKCV*. https://figshare.com/s/1fc592780ab44ccfa39e

[2] 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.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Springer.

[4] 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.

[5] Anoud Alshnakat, Dilian Gurov, Christian Lidström, and Philipp Rümmer. 2020. Constraint-Based Contract Inference for Deductive Verification. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*. 149–176.

[6] 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.

[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, François-Xavier Standaert, and Pierre-Yves Strub. 2020. Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations. *J. Cryptogr. Eng.* 10, 1 (2020), 17–26.

[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, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. 2021. Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 2 (2021), 189–228.

[13] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference*. 230–235.

[14] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. 2016. Randomness Complexity of Private Circuits for Multiplication. In *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 616–648.

[15] 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.

[16] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. 2020. Random Probing Security: Verification, Composition, Expansion and New Constructions. In *Proceedings of the 40th Annual International Cryptology Conference*. 339–368.

[17] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. 2020. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 311–341.

[18] 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.

[19] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. 2022. IronMask: Versatile Verification of Masking Security. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*. 142–160.

[20] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. 2017. Optimal first-order boolean masking for embedded iot devices. In *Proceedings of the International Conference on Smart Card Research and Advanced Applications*. 22–41.

[21] 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.

[22] Sandrine Blazy, David Pichardie, and Alix Trieu. 2019. Verifying constant-time implementations by abstract interpretation. *J. Comput. Secur.* (2019).

[23] 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.

[24] 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.

[25] 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.

[26] Nicolas Bordes and Pierre Karpman. 2021. Fast verification of masking schemes in characteristic two. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 283–312.

[27] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[28] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM fuzzing for JIT-induced side-channel detection. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 1011–1023.

[29] 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.

[30] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. 2020. Hardware Private Circuits: From Trivial Composition to Full Verification. *IACR Cryptol. ePrint Arch.* (2020), 185.

[31] Gaëtan Cassiers and François-Xavier Standaert. 2020. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Inf. Forensics Secur.* (2020), 2542–2555.

[32] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 913–926.

[33] 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.

[34] Yueqiang Cheng, Zhi Zhang, Yansong Gao, Zhaofeng Chen, Shengjian Guo, Qifei Zhang, Rui Mei, Surya Nepal, and Yang Xiang. 2022. Meltdown-type attacks are still feasible in the wall of kernel page-Table isolation. *Comput. Secur.* 113 (2022), 102556.

[35] 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.

[36] Jean-Sébastien Coron. 1999. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*. 292–302.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *Proceedings of 2020 IEEE Symposium on Security and Privacy*.

[43] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium*.

[44] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[45] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* (2015).

[46] 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.

[47] 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.

[48] 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.

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

[50] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. 2015. Quantitative Masking Strength: Quantifying the Power Side-Channel Resistance of Software Code. *IEEE Transactions on CAD of Integrated Circuits and Systems* 34, 10 (2015), 1558–1568.

[51] 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.

[52] 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 (2021), 26:1–26:42.

[53] Pengfei Gao, Hongyi Xie, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. 2022. Formal Verification of Masking Countermeasures for Arithmetic Programs. *IEEE Trans. Software Eng.* 48, 3 (2022), 973–1000. https://doi.org/10.1109/

TSE.2020.3008852

[54] 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.

[55] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and Quantifying Side-channel Resistance of Masked Software Implementations. *ACM Trans. Softw. Eng. Methodol.* 28, 3 (2019), 16:1–16:32. https://doi.org/10.1145/3330392

[56] Louis Goubin and Jacques Patarin. 1999. DES and Differential Power Analysis (The "Duplication" Method). In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*. 158–172.

[57] Dahmun Goudarzi and Matthieu Rivain. 2017. How Fast Can Higher-Order Masking Be in Software?. In *Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 567–597.

[58] Hannes Groß and Stefan Mangard. 2018. A Unified Masking Approach. *Journal of Cryptographic Engineering* 8, 2 (2018), 109–124.

[59] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: speculative symbolic execution for cache timing leak detection. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 1235–1247.

[60] Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. 2020. Exposing cache timing side-channel leaks through out-of-order symbolic execution. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 147:1–147:32.

[61] 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.

[62] Reiner Hähnle, Ina Schaefer, and Richard Bubel. 2013. Reuse in Software Verification by Abstract Method Calls. In *Proceedings of the 24th International Conference on Automated Deduction*. 300–314.

[63] 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.

[64] 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.

[65] Ismet Burak Kadron and Tevfik Bultan. 2022. TSA: a tool to detect and quantify network side-channels. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1760–1764.

[66] Ismet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-driven side-channel analysis for networked applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 260–271.

[67] 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.

[68] Pierre Karpman and Daniel S Roche. 2018. New instantiations of the CRYPTO 2017 masking schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*. 285–314.

[69] David Knichel, Pascal Sasdrich, and Amir Moradi. 2020. SILVER - Statistical Independence and Leakage Verification. In *Proceedings of the 26th International Conference on the Theory and Application of Cryptology and Information Security*. 787–816.

[70] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[71] 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.

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

[73] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium*.

[74] 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.

[75] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *Proceedings of the 20th European Symposium on Research in Computer Security*.

[76] Quentin L Meunier, Inès Ben El Ouahma, and Karine Heydemann. 2020. SELA: a Symbolic Expression Leakage Analyzer. In *Proceedings of the International Workshop on Security Proofs for Embedded Systems*.

[77] Quentin L. Meunier, Etienne Pons, and Karine Heydemann. 2021. LeakageVerif: Scalable and Efficient Leakage Verification in Symbolic Expressions. Cryptology ePrint Archive, Report 2021/1468.

[78] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. 2019. Glitch-Resistant Masking Revisited or Why Proofs in the Robust Probing Model are Needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 2 (2019), 256–292.

[79] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DifFuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering*.

[80] 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.

[81] 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*.

[82] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Proceedings of the IEEE 29th Computer Security Foundations Symposium*.

[83] 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.

[84] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. 2009. Statistical Analysis of Second Order Differential Power Analysis. *IEEE Trans. Computers* 58, 6 (2009), 799–811.

[85] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. 2022. DeJITLeak: eliminating JIT-induced timing side-channel leaks. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 872–884. https://doi.org/10.1145/3540250.3549150

[86] 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 Cryptol. ePrint Arch.* 2019 (2019), 948.

[87] 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.

[88] Nicolás Rosner, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2019. Profit: Detecting and Quantifying Side Channels in Networked Applications. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*.

[89] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. 2020. A Power Side-Channel Attack on the CCA2-Secure HQC KEM. In *Proceedings of the 19th International Conference on Smart Card Research and Advanced Applications*. 119–134.

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

[91] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 810–822.

[92] 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.

[93] Weijia Wang, Chun Guo, François-Xavier Standaert, Yu Yu, and Gaëtan Cassiers. 2020. Packed Multiplication: How to Amortize the Cost of Side-Channel Masking?. In *International Conference on the Theory and Application of Cryptology and Information Security*. 851–880.

[94] 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 Trans. Information Forensics and Security* 13, 5 (2018), 1301–1316.

[95] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.* (2019).

[96] 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.

[97] 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.

[98] 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.

A APPENDIX

A.1 The type system of Barthe et al. [9]

We explain why the type system of Barthe et al. [9] fails on XORMULTI.

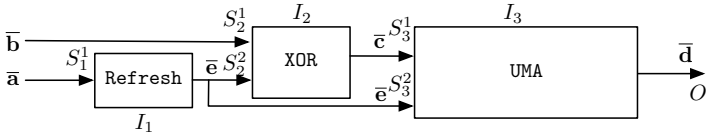

Fig. 8. Input-output relation of XORMULTI

Fig. 8 shows the input-output relation of XORMULTI. It can easy to conclude that Refresh is 1-SNI, XOR is 1-NI and UMA is 1-SNI. Let O denote the set of observed variable from the output of XORMULTI. Let I_i denote the set of observed internal variables from i^{th} gadget call. Let S_i^j denote the number of input shares needed to simulate further internal nodes. To prove whehter XORMULTI is 1-NI, The global constraint is $|I_1| + |I_2| + |I_3| + |O| \leq 1$. Let us get started from right to left in Fig. 8.

(1) The side condition $|O| + |I_3| \leq 1$ is satisfied. As UMA is 1-SNI, we can obtain $|S_3^1| \leq |I_3|$ and $|S_3^2| \leq |I_3|$.
(2) The side condition $|S_3^1| + |I_2| \leq 1$ is satisfied. As XOR is 1-NI, we can obtain $|S_2^1| \leq |I_2| + |S_3^1|$ and $|S_2^2| \leq |I_2| + |S_3^1|$.
(3) The side condition $|I_1| + |S_3^2| + |S_2^2| \leq 1$ cannot be proved through the constraints as we can only conclude that $|I_1| + |S_3^2| + |S_2^2| \leq |I_1| + 2|I_3| + |I_2|$. The proof terminates here.

From above, it is easy to conclude that Barthe's work can prove neither 1-NI nor 1-SNI of XORMULTI.