Automated Verification of Correctness for Masked Arithmetic Programs

Mingyang Liu[1], Fu Song[1,2,3(✉)], and Taolue Chen[4]

[1] ShanghaiTech University, Shanghai 201210, China
[2] Institute of Software, Chinese Academy of Sciences & University of Chinese Academy of Sciences, Beijing 100190, China
songfu@shanghaitech.edu.cn
[3] Automotive Software Innovation Center, Chongqing 400000, China
[4] Birkbeck, University of London, London WC1E 7HX, UK

Abstract. Masking is a widely-used effective countermeasure against power side-channel attacks for implementing cryptographic algorithms. Surprisingly, few formal verification techniques have addressed a fundamental question, i.e., whether the masked program and the original (unmasked) cryptographic algorithm are functional equivalent. In this paper, we study this problem for masked arithmetic programs over Galois fields of characteristic 2. We propose an automated approach based on term rewriting, aided by random testing and SMT solving. The overall approach is sound, and complete under certain conditions which do meet in practice. We implement the approach as a new tool FISCHER and carry out extensive experiments on various benchmarks. The results confirm the effectiveness, efficiency and scalability of our approach. Almost all the benchmarks can be proved for the first time by the term rewriting system solely. In particular, FISCHER detects a new flaw in a masked implementation published in EUROCRYPT 2017.

1 Introduction

Power side-channel attacks [42] can infer secrecy by statistically analyzing the power consumption during the execution of cryptographic programs. The victims include implementations of almost all major cryptographic algorithms, e.g., DES [41], AES [54], RSA [33], Elliptic curve cryptography [46,52] and post-quantum cryptography [56,59]. To mitigate the threat, cryptographic algorithms are often implemented via *masking* [37], which divides each secret value into $(d+1)$ shares by randomization, where d is a given masking order. However, it is error-prone to implement secure and correct masked implementations for non-linear functions (e.g., finite-field multiplication, module addition and S-Box),

This work is supported by the National Natural Science Foundation of China (62072309), CAS Project for Young Scientists in Basic Research (YSBR-040), ISCAS New Cultivation Project (ISCAS-PYFX-202201), an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), and Birkbeck BEI School Project (EFFECT).

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 255–280, 2023.
https://doi.org/10.1007/978-3-031-37709-9_13

which are prevalent in cryptography. Indeed, published implementations of AES S-Box that have been proved secure via paper-and-pencil [19,40,58] were later shown to be vulnerable to power side-channels when d is no less than 4 [24].

While numerous formal verification techniques have been proposed to prove resistance of masked cryptographic programs against power side-channel attacks (e.g., [7,13,26,29–32,64]), one fundamental question which is largely left open is the (functional) correctness of the masked cryptographic programs, i.e., whether a masked program and the original (unmasked) cryptographic algorithm are actually functional equivalent. It is conceivable to apply general-purpose program verifiers to masked cryptographic programs. Constraint-solving based approaches are available, for instance, Boogie [6] generates constraints via weakest precondition reasoning which then invokes SMT solvers; SeaHorn [36] and CPAChecker [12] adopt model checking by utilizing SMT or CHC solvers. More recent work (e.g., CryptoLine [28,45,53,62]) resorts to computer algebra, e.g., to reduce the problem to the ideal membership problem. The main challenge of applying these techniques to masked cryptographic programs lies in the presence of finite-field multiplication, affine transformations and bitwise exclusive-OR (XOR). For instance, finite-field multiplication is not natively supported by the current SMT or CHC solvers, and the increasing number of bitwise XOR operations causes the infamous state-explosion problem. Moreover, to the best of our knowledge, current computer algebra systems do not provide the full support required by verification of masked cryptographic programs.

Contributions. We propose a novel, term rewriting based approach to efficiently check whether a masked program and the original (unmasked) cryptographic algorithm (over Galois fields of characteristic 2) are functional equivalent. Namely, we provide a term rewriting system (TRS) which can handle affine transformations, bitwise XOR, and finite-field multiplication. The verification problem is reduced to checking whether a term can be rewritten to normal form 0. This approach is sound, i.e., once we obtain 0, we can claim functional equivalence. In case the TRS reduces to a normal form which is different from 0, most likely they are *not* functional equivalent, but a false positive is possible. We further resort to random testing and SMT solving by directly analyzing the obtained normal form. As a result, it turns out that the overall approach is complete if no uninterpreted functions are involved in the normal form.

We implement our approach as a new tool FISCHER (**F**unctional**I**ty of ma**S**ked **C**ryptograp**H**ic program verifi**ER**), based on the LLVM framework [43]. We conduct extensive experiments on various masked cryptographic program benchmarks. The results show that our term rewriting system solely is able to prove almost all the benchmarks. FISCHER is also considerably more efficient than the general-purpose verifiers SMACK [55], SeaHorn, CPAChecker, and Symbiotic [22], cryptography-specific verifier CryptoLine, as well as a straightforward approach that directly reduces the verification task to SMT solving. For instance, our approach is able to handle masked implementations of finite-field multiplication with masking orders up to 100 in less than 153 s, while none of the compared approaches can handle masking order of 3 in 20 min.

In particular, for the first time we detect a flaw in a masked implementation of finite-field multiplication published in EUROCRYPT 2017 [8]. The flaw is tricky, as it only occurs for the masking order $d \equiv 1 \mod 4$.[1] This finding highlights the importance of the correctness verification of masked programs, which has been largely overlooked, but of which our work provides an effective solution.

Our main contributions can be summarized as follows.

- We propose a term rewriting system for automatically proving the functional correctness of masked cryptographic programs;
- We implement a tool FISCHER by synergistically integrating the term rewriting based approach, random testing and SMT solving;
- We conduct extensive experiments, confirming the effectiveness, efficiency, scalability and applicability of our approach.

Related Work. Program verification has been extensively studied for decades. Here we mainly focus on their application in cryptographic programs, for which some general-purpose program verifiers have been adopted. Early work [3] uses Boogie [6]. HACL* [65] uses F* [2] which verifies programs by a combination of SMT solving and interactive proof assistants. Vale [15] uses F* and Dafny [44] where Dafny harnesses Boogie for verification. Cryptol [61] checks equivalence between machine-readable cryptographic specifications and real-world implementations via SMT solving. As mentioned before, computer algebra systems (CAS) have also been used for verifying cryptographic programs and arithmetic circuits, by reducing to the ideal membership problem together with SAT/SMT solving. Typical work includes CryptoLine and AMulet [38,39]. However, as shown in Sect. 7.2, neither general-purpose verifiers (SMACK with Boogie and Corral, SeaHorn, CPAChecker and Symbiotic) nor the CAS-based verifier CryptoLine is sufficiently powerful to verify masked cryptographic programs. Interactive proof assistants (possibly coupled with SMT solvers) have also been used to verify unmasked cryptographic programs (e.g., [1,4,9,23,27,48,49]). Compared to them, our approach is highly automatic, which is more acceptable and easier to use for general software developers.

Outline. Section 2 recaps preliminaries. Section 3 presents a language on which the cryptographic program is formalized. Section 4 gives an example and an overview of our approach. Section 5 and Sect. 6 introduce the term rewriting system and verification algorithms. Section 7 reports experimental results. We conclude in Sect. 8. The source code of our tool and benchmarks are available at https://github.com/S3L-official/FISCHER.

2 Preliminaries

For two integers l, u with $l \leq u$, $[l, u]$ denotes the set of integers $\{l, l+1, \cdots, u\}$.

Galois Field. A *Galois field* $\mathbb{GF}(p^n)$ comprises polynomials $a_{n-1}X^{n-1} + \cdots + a_1X^1 + a_0$ over $\mathbb{Z}_p = [0, p-1]$, where p is a prime number, n is a positive integer, and $a_i \in \mathbb{Z}_p$. (Here p is the *characteristic* of the field, and p^n

[1] This flaw has been confirmed by an author of [8].

is the *order* of the field.) Symmetric cryptography (e.g., DES [50], AES [25], SKINNY [10], PRESENT [14]) and bitsliced implementations of asymmetric cryptography (e.g., [17]) intensively uses $\mathbb{GF}(2^n)$. Throughout the paper, $\mathbb{F}$ denotes the Galois field $\mathbb{GF}(2^n)$ for a fixed n, and $\oplus$ and $\otimes$ denote the addition and multiplication on $\mathbb{F}$, respectively. Recall that $\mathbb{GF}(2^n)$ can be constructed from the quotient ring of the polynomial ring $\mathbb{GF}(2)[X]$ with respect to the ideal generated by an irreducible polynomial P of degree n. Hence, multiplication is the product of two polynomials modulo P in $\mathbb{GF}(2)[X]$ and addition is bitwise exclusive-OR (XOR) over the binary representation of polynomials. For example, AES uses $\mathbb{GF}(256) = \mathbb{GF}(2)[X]/(X^8 + X^4 + X^3 + X + 1)$. Here $n = 8$ and $P = X^8 + X^4 + X^3 + X + 1$.

Higher-Order Masking. To achieve order-d security against power side-channel attacks under certain leakage models, masking is usually used [37,60]. Essentially, masking partitions each secret value into (usually $d + 1$) shares so that knowing at most d shares cannot infer any information of the secret value, called *order-d masking*. In Boolean masking, a value $a \in \mathbb{F}$ is divided into shares $a_0, a_1, \ldots, a_d \in \mathbb{F}$ such that $a_0 \oplus a_1 \oplus \ldots \oplus a_d = a$. Typically, $a_1, \ldots, a_d$ are random values and $a_0 = a \oplus a_1 \oplus \ldots \oplus a_d$. The tuple $(a_0, a_1, \ldots, a_d)$, denoted by $\mathbf{a}$, is called an *encoding* of a. We write $\bigoplus_{i \in [0,d]} \mathbf{a}_i$ (or simply $\bigoplus \mathbf{a}$) for $a_0 \oplus a_1 \oplus \ldots \oplus a_d$. Additive masking can be defined similarly to Boolean masking, where $\oplus$ is replaced by the module arithmetic addition operator. In this work, we focus on Boolean masking as the XOR operation is more efficient to implement.

To implement a masked program, for each operation in the cryptographic algorithm, a corresponding operation on shares is required. As we will see later, when the operation is affine (i.e. the operation f satisfies $f(x \oplus y) = f(x) \oplus f(y) \oplus c$ for some constant c), the corresponding operation is simply to apply the original operation on each share a_i in the encoding $(a_0, a_1, \ldots, a_d)$. However, for non-affine operations (e.g., multiplication and addition), it is a very difficult task and error-prone [24]. Ishai et al. [37] proposed the first masked implementation of multiplication, but limited to the domain $\mathbb{GF}(2)$ *only*. The number of the required random values and operations is not optimal and is known to be vulnerable in the presence of glitches because the electric signals propagate at different speeds in the combinatorial paths of hardware circuits. Thus, various follow-up papers proposed ways to implement higher-order masking for the domain $\mathbb{GF}(2^n)$ and/or optimizing the computational complexity, e.g., [8,11,21,34,58], all of which are referred to as ISW scheme in this paper. In another research direction, new glitch-resistant Boolean masking schemes have been proposed, e.g., Hardware Private Circuits (HPC1 & HPC2) [20], Domain-oriented Masking (DOM) [35] and Consolidating Masking Schemes (CMS) [57]. In this work, we are interested in automatically proving the correctness of the masked programs.

3 The Core Language

In this section, we first present the core language MSL, given in Fig. 1, based on which the verification problem is formalized.

$\langle \text{expr} \rangle ::= \langle \text{var} \rangle \mid \langle \text{num} \rangle \mid \langle \text{expr} \rangle \oplus \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \otimes \langle \text{expr} \rangle \mid (\langle \text{expr} \rangle)$

$\langle \text{stmts} \rangle ::= \langle \text{var} \rangle \leftarrow \langle \text{expr} \rangle \mid \langle \text{var} \rangle \leftarrow \texttt{rand} \mid \langle \text{stmts} \rangle \ \langle \text{stmts} \rangle$

$\langle \text{var} \rangle \leftarrow \langle \text{id} \rangle_{\text{proc}} (\langle \text{vars} \rangle) \mid \langle \text{var} \rangle \leftarrow \langle \text{id} \rangle_{\text{affine}} (\langle \text{var} \rangle)$

$\langle \text{proc} \rangle ::= \texttt{proc} \langle \text{id} \rangle \ \texttt{input} \langle \text{ids} \rangle \ \texttt{output} \langle \text{id} \rangle \ \langle \text{stmts} \rangle_{\text{origin}} \ \texttt{shares} \ \langle \text{num} \rangle \ \langle \text{stmts} \rangle_{\text{masked}}$

$\langle \text{affine} \rangle ::= \texttt{affine} \langle \text{id} \rangle \ [\texttt{input} \langle \text{id} \rangle \ \texttt{output} \langle \text{id} \rangle \ \langle \text{stmts} \rangle]$

Fig. 1. Syntax of MSL in Backus-Naur form

A program $\mathcal{P}$ in MSL is given by a sequence of procedure definitions and affine transformation definitions/declarations. A procedure definition starts with the keyword proc, followed by a procedure name, a list of input parameters, an output and its body. The procedure body has two blocks of statements, separated by a special statement shares $d+1$, where d is the masking order. The first block $\langle \text{stmts} \rangle_{\text{origin}}$, called the *original block*, implements its original functionality on the input parameters without masking. The second block $\langle \text{stmts} \rangle_{\text{masked}}$, called the *masked block*, is a masked implementation of the original block over the input encodings $\mathbf{x}$ of the input parameters x. The input parameters and output x, declared using the keywords input and output respectively, are scalar variables in the original block, but are treated as the corresponding encodings (i.e., tuples) $\mathbf{x}$ in the masked block. For example, input x declares the scalar variable x as the input of the original block, while it implicitly declares an encoding $\mathbf{x} = (x_0, x_1, \ldots, x_d)$ as the input of the masked block with shares $d+1$.

We distinguish affine transformation definitions and declarations. The former starts with the keyword affine, followed by a name f, an input, an output and its body. It is expected that the affine property $\forall x, y \in \mathbb{F}. f(x \oplus y) = f(x) \oplus f(y) \oplus c$ holds for some affine constant $c \in \mathbb{F}$. (Note that the constant c is not explicitly provided in the program, but can be derived, cf. Sect. 6.2.) The transformation f is *linear* if its affine constant c is 0. In contrast, an affine transformation declaration f simply declares a transformation. As a result, it can only be used to declare a linear one (i.e., c must be 0), which is treated as an uninterpreted function. Note that non-linear affine transformation declarations can be achieved by declaring linear affine transformations and affine transformation definitions. Affine transformation here serves as an abstraction to capture complicated operations (e.g., shift, rotation and bitwise Boolean operations) and can accelerate verification by expressing operations as uninterpreted functions. In practice, a majority of cryptographic algorithms (in symmetric cryptography) can be represented by a composition of S-box, XOR and linear transformation only.

Masking an affine transformation can simply mask an input encoding in a share-wise way, namely, the masked version of the affine transformation $f(a)$ is

$$f(a_0 \oplus a_1 \oplus \ldots \oplus a_d) = \begin{cases} f(a_0) \oplus f(a_1) \oplus \ldots \oplus f(a_d), & \text{if } d \text{ is even;} \\ f(a_0) \oplus f(a_1) \oplus \ldots \oplus f(a_d) \oplus c, & \text{if } d \text{ is odd.} \end{cases}$$

This is default, so affine transformation definition only contains the original block but no masked block.

A statement is either an assignment or a function call. MSL features two types of assignments which are either of the form $x \leftarrow e$ defined as usual or of the form $r \leftarrow$rand which assigns a uniformly sampled value from the domain $\mathbb{F}$ to the variable r. As a result, r should be read as a random variable. We assume that each random variable is defined only once. We note that the actual parameters and output are scalar if the procedure is invoked in an original block while they are the corresponding encodings if it is invoked in a masked block.

MSL is the core language of our tool. In practice, to be more user-friendly, our tool also accepts C programs with conditional branches and loops, both of which should be statically determinized (e.g., loops are bound and can be unrolled; the branching of conditionals can also be fixed after loop unrolling). Furthermore, we assume there is no recursion and dynamic memory allocation. These restrictions are sufficient for most symmetric cryptography and bitsliced implementations of public-key cryptography, which mostly have simple control graphs and memory aliases.

Problem Formalization. Fix a program $\mathcal{P}$ with all the procedures using order-d masking. We denote by $\mathcal{P}_{\mathfrak{o}}$ (resp. $\mathcal{P}_{\mathfrak{m}}$) the program $\mathcal{P}$ where all the masked (resp. original) blocks are omitted. For each procedure f, the procedures $f_{\mathfrak{o}}$ and $f_{\mathfrak{m}}$ are defined accordingly.

Definition 1. *Given a procedure f of $\mathcal{P}$ with m input parameters, $f_{\mathfrak{m}}$ and $f_{\mathfrak{o}}$ are* functional equivalent*, denoted by $f_{\mathfrak{m}} \cong f_{\mathfrak{o}}$, if the following statement holds:*

$$\forall a^1, \cdots, a^m, r_1, \cdots, r_h \in \mathbb{F}, \forall \mathbf{a}^1, \cdots, \mathbf{a}^m \in \mathbb{F}^{d+1}.$$
$$\left(\bigwedge_{i\in[1,m]} a^i = \bigoplus_{j\in[0,d]} \mathbf{a}^i_j\right) \rightarrow \left(f_{\mathfrak{o}}(a^1, \cdots, a^m) = \bigoplus_{i\in[0,d]} f_{\mathfrak{m}}(\mathbf{a}^1, \cdots, \mathbf{a}^m)_i\right)$$

where $r_1, \cdots, r_h$ are all the random variables used in $f_{\mathfrak{m}}$.

Note that although the procedure $f_{\mathfrak{m}}$ is randomized (i.e., the output encoding $f_{\mathfrak{m}}(\mathbf{a}^1, \cdots, \mathbf{a}^m_i)$ is technically a random variable), for functional equivalence we consider a stronger notion, viz., to require that $f_{\mathfrak{m}}$ and $f_{\mathfrak{o}}$ are equivalent under any values in the support of the random variables $r_1, \cdots, r_h$. Thus, $r_1, \cdots, r_h$ are universally quantified in Definition 1.

The verification problem is to check if $f_{\mathfrak{m}} \cong f_{\mathfrak{o}}$ for a given procedure f where $\bigwedge_{i\in[1,m]} a^i = \bigoplus_{j\in[0,d]} \mathbf{a}^i_j$ and $f_{\mathfrak{o}}(a^1, \cdots, a^m) = \bigoplus_{i\in[0,d]} f_{\mathfrak{m}}(\mathbf{a}^1, \cdots, \mathbf{a}^m)_i$ are regarded as pre- and post-conditions, respectively. Thus, we assume the unmasked procedures themselves are correct (which can be verified by, e.g., CryptoLine). Our focus is on whether the masked counterparts are functional equivalent to them.

4 Overview of the Approach

In this section, we first present a motivating example given in Fig. 2, which computes the multiplicative inverse in $\mathbb{GF}(2^8)$ for the AES S-Box [58] using first-order

```
affine exp2 input x output y
  y ← x ⊗ x
affine exp4 input x output y
  y ← exp2(exp2(x))
affine exp16 input x output y
  y ← exp4(exp4(x))

proc sec_mult input a b output c
  c ← a ⊗ b
shares 2
  r0 ← rand
  r1 ← r0 ⊕ (a0 ⊗ b1) ⊕ (a1 ⊗ b0)
  c0 ← (a0 ⊗ b0) ⊕ r0
  c1 ← (a1 ⊗ b1) ⊕ r1

proc refresh_masks input x output y
  y ← x
shares 2
  r0 ← rand
  y0 ← x0 ⊕ r0   y1 ← x1 ⊕ r0
```

```
proc sec_exp254 input x output y
  z ← exp2(x)
  y ← z ⊗ x
  w ← exp4(y)
  y ← y ⊗ w
  y ← exp16(y)
  y ← y ⊗ w
  y ← y ⊗ z
shares 2
  z0 ← exp2(x0)
  z1 ← exp2(x1)
  z⃗ ← refresh_masks(z⃗)
  y⃗ ← sec_mult(z⃗, x⃗)
  w0 ← exp4(y0)
  w1 ← exp4(y1)
  w⃗ ← refresh_masks(w⃗)
  y⃗ ← sec_mult(y⃗, w⃗)
  y0 ← exp16(y0)
  y1 ← exp16(y1)
  y⃗ ← sec_mult(y⃗, w⃗)
  y⃗ ← sec_mult(y⃗, z⃗)
```

Fig. 2. Motivating example, where $\mathbf{x}$ denotes (x_0, x_1).

Boolean masking. It consists of three affine transformation definitions and two procedure definitions. For a given input x, `exp2`(x) outputs x^2, `exp4`(x) outputs x^4 and `exp16`(x) outputs x^{16}. Obviously, these three affine transformations are indeed linear.

Procedure $\texttt{sec\_mult}_\mathtt{o}(a, b)$ outputs $a \otimes b$. Its masked version $\texttt{sec\_mult}_\mathtt{m}(\mathbf{a}, \mathbf{b})$ computes the encoding $\mathbf{c} = (c_0, c_1)$ over the encodings $\mathbf{a} = (a_0, a_1)$ and $\mathbf{b} = (b_0, b_1)$. Clearly, it is desired that $c_0 \oplus c_1 = a \otimes b$ if $a_0 \oplus a_1 = a$ and $b_0 \oplus b_1 = b$. Procedure $\texttt{refresh\_masks}_\mathtt{o}(x)$ is the identity function while its masked version $\texttt{refresh\_masks}_\mathtt{m}(\mathbf{x})$ re-masks the encoding $\mathbf{x}$ using a random variable r_0. Thus, it is desired that $y_0 \oplus y_1 = x$ if $x = x_0 \oplus x_1$. Procedure $\texttt{sec\_exp254}_\mathtt{o}(x)$ computes the multiplicative inverse x^{254} of x in $\mathbb{GF}(2^8)$. Its masked version $\texttt{sec\_exp254}_\mathtt{m}(\mathbf{x})$ computes the encoding $\mathbf{y} = (y_0, y_1)$ where $\texttt{refresh\_masks}_\mathtt{m}$ is invoked to avoid power side-channel leakage. Thus, it is desired that $y_0 \oplus y_1 = x^{254}$ if $x_0 \oplus x_1 = x$. In summary, it is required to prove $\texttt{sec\_mult}_\mathtt{m} \cong \texttt{sec\_mult}_\mathtt{o}$, $\texttt{refresh\_masks}_\mathtt{m} \cong \texttt{refresh\_masks}_\mathtt{o}$ and $\texttt{sec\_exp254}_\mathtt{m} \cong \texttt{sec\_exp254}_\mathtt{o}$.

4.1 Our Approach

An overview of FISCHER is shown in Fig. 3. The input program is expected to follow the syntax of MSL but in C language. Moreover, the pre-conditions and post-conditions of the verification problem are expressed by assume and assert statements in the masked procedure, respectively. Recall that the input program can contain conditional branches and loops when are statically determinized. Furthermore, affine transformations can use other common operations (e.g., shift, rotation and bitwise Boolean operations) besides the addition $\oplus$ and multiplication $\otimes$ on the underlying field $\mathbb{F}$. FISCHER leverages the LLVM framework to obtain the LLVM intermediate representation (IR) and call graph, where

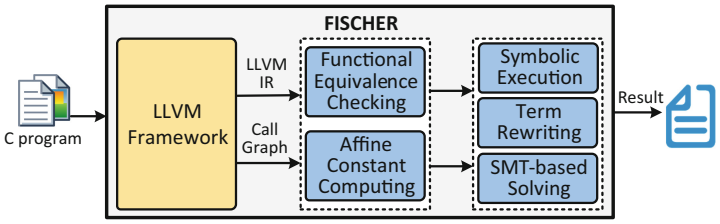

Fig. 3. Overview of FISCHER.

all the procedure calls are inlined. It then invokes *Affine Constant Computing* to iteratively compute the affine constants for affine transformations according to the call graph, and *Functional Equivalence Checking* to check functional equivalence, both of which rely on the underpinning engines, viz., *Symbolic Execution* (refer to symbolic computation without path constraint solving in this work), *Term Rewriting* and *SMT-based Solving*.

We apply intra-procedural symbolic execution to compute the symbolic outputs of the procedures and transformations, i.e., expressions in terms of inputs, random variables and affine transformations. The symbolic outputs are treated as terms based on which both the problems of functional equivalence checking and affine constant computing are solved by rewriting to their normal forms (i.e., sums of monomials w.r.t. a total order). The analysis result is often conclusive from normal forms. In case it is inconclusive, we iteratively inline affine transformations when their definitions are available until either the analysis result is conclusive or no more affine transformations can be inlined. If the analysis result is still inconclusive, to reduce false positives, we apply random testing and accurate (but computationally expansive) SMT solving to the normal forms instead of the original terms. We remark that the term rewriting system solely can prove almost all the benchmarks in our experiments.

Consider the motivating example. To find the constant $c \in \mathbb{F}$ of `exp2` such that the property $\forall x, y \in \mathbb{F}.\texttt{exp2}(x \oplus y) = \texttt{exp2}(x) \oplus \texttt{exp2}(y) \oplus c$ holds, by applying symbolic execution, $\texttt{exp2}(x)$ is expressed as the term $x \otimes x$. Thus, the property is reformulated as $(x \oplus y) \otimes (x \oplus y) = (x \otimes x) \oplus (y \otimes y) \oplus c$, from which we can deduce that the desired affine constant c is equivalent to the term $((x \oplus y) \otimes (x \oplus y)) \oplus (x \otimes x) \oplus (y \otimes y)$. Our TRS will reduce the term as follows:

$$
\begin{aligned}
& \underline{((x \oplus y) \otimes (x \oplus y))} \oplus (x \otimes x) \oplus (y \otimes y) && \text{Distributive Law} \\
= {} & \underline{(x \otimes (x \oplus y))} \oplus \underline{(y \otimes (x \oplus y))} \oplus (x \otimes x) \oplus (y \otimes y) && \text{Distributive Law} \\
= {} & (x \otimes x) \oplus (x \otimes y) \oplus \underline{(y \otimes x)} \oplus (y \otimes y) \oplus (x \otimes x) \oplus (y \otimes y) && \text{Commutative Law} \\
= {} & (x \otimes x) \oplus (x \otimes y) \oplus \underline{(x \otimes y)} \oplus (y \otimes y) \oplus \underline{(x \otimes x)} \oplus (y \otimes y) && \text{Commutative Law} \\
= {} & \underline{(x \otimes x) \oplus (x \otimes x)} \oplus \underline{(x \otimes y) \oplus (x \otimes y)} \oplus \underline{(y \otimes y) \oplus (y \otimes y)} = 0 && \text{Zero Law of XOR}
\end{aligned}
$$

For the transformation $\texttt{exp4}(x)$, by applying symbolic execution, it can be expressed as the term $\texttt{exp2}(\texttt{exp2}(x))$. To find the constant $c \in \mathbb{F}$ to satisfy $\forall x, y \in \mathbb{F}.\texttt{exp4}(x \oplus y) = \texttt{exp4}(x) \oplus \texttt{exp4}(y) \oplus c$, we compute the term $\texttt{exp2}(\texttt{exp2}(x \oplus y)) \oplus \texttt{exp2}(\texttt{exp2}(x)) \oplus \texttt{exp2}(\texttt{exp2}(y))$. By applying our TRS, we have:

$$\begin{aligned}
&\underline{\texttt{exp2}(\texttt{exp2}(x \oplus y))} \oplus \texttt{exp2}(\texttt{exp2}(x)) \oplus \texttt{exp2}(\texttt{exp2}(y)) \\
&= \underline{\texttt{exp2}(\texttt{exp2}(x) \oplus \texttt{exp2}(y))} \oplus \texttt{exp2}(\texttt{exp2}(x)) \oplus \texttt{exp2}(\texttt{exp2}(y)) \\
&= \texttt{exp2}(\texttt{exp2}(x)) \oplus \texttt{exp2}(\texttt{exp2}(y)) \oplus \underline{\texttt{exp2}(\texttt{exp2}(x))} \oplus \texttt{exp2}(\texttt{exp2}(y)) \\
&= \underline{\texttt{exp2}(\texttt{exp2}(x)) \oplus \texttt{exp2}(\texttt{exp2}(x))} \oplus \underline{\texttt{exp2}(\texttt{exp2}(y)) \oplus \texttt{exp2}(\texttt{exp2}(y))} = 0
\end{aligned}$$

Clearly, the affine constant of `exp4` is 0. Similarly, we can deduce that the affine constant of the transformation `exp16` is 0 as well.

To prove $\texttt{sec\_mult}_\texttt{o} \cong \texttt{sec\_mult}_\texttt{m}$, by applying symbolic execution, we have that $\texttt{sec\_mult}_\texttt{o}(a, b) = a \otimes b$ and $\texttt{sec\_mult}_\texttt{m}(\mathbf{a}, \mathbf{b}) = \mathbf{c} = (c_0, c_1)$, where $c_0 = (a_0 \otimes b_0) \oplus r_0$ and $c_1 = (a_1 \otimes b_1) \oplus (r_0 \oplus (a_0 \otimes b_1) \oplus (a_1 \otimes b_0))$. Then, by Definition 1, it suffices to check

$$\begin{aligned}
\forall a, b, a_0, a_1, b_0, b_1, r_0 \in \mathbb{F}. \big(a = a_0 \oplus a_1 \wedge b = b_0 \oplus b_1\big) \rightarrow \\
\big(a \otimes b = ((a_0 \otimes b_0) \oplus r_0) \oplus \big((a_1 \otimes b_1) \oplus (r_0 \oplus (a_0 \otimes b_1) \oplus (a_1 \otimes b_0))\big)\big).
\end{aligned}$$

Thus, we check the term $\big((a_0 \oplus a_1) \otimes (b_0 \oplus b_1)\big) \oplus ((a_0 \otimes b_0) \oplus r_0) \oplus ((a_1 \otimes b_1) \oplus (r_0 \oplus (a_0 \otimes b_1) \oplus (a_1 \otimes b_0)))$ which is equivalent to 0 iff $\texttt{sec\_mult}_\texttt{o} \cong \texttt{sec\_mult}_\texttt{m}$. Our TRS is able to reduce the term to 0. Similarly, we represent the outputs of $\texttt{sec\_exp254}_\texttt{o}$ and $\texttt{sec\_exp254}_\texttt{m}$ as terms via symbolic execution, from which the statement $\texttt{sec\_exp254}_\texttt{o} \cong \texttt{sec\_exp254}_\texttt{m}$ is also encoded as a term, which can be reduced to 0 via our TRS without inlining any transformations.

5 Term Rewriting System

In this section, we first introduce some basic notations and then present our term rewriting system.

Definition 2. *Given a program $\mathcal{P}$ over $\mathbb{F}$, a* signature $\Sigma_\mathcal{P}$ *of $\mathcal{P}$ is a set of symbols $\mathbb{F} \cup \{\oplus, \otimes, f_1, \ldots, f_t\}$, where $s \in \mathbb{F}$ with arity 0 are all the constants in $\mathbb{F}$, $\oplus$ and $\otimes$ with arity 2 are addition and multiplication operators on $\mathbb{F}$, and $f_1, \cdots, f_t$ with arity 1 are affine transformations defined/declared in $\mathcal{P}$.*

For example, the signature of the motivating example is $\mathbb{F} \cup \{\oplus, \otimes, \texttt{exp2}, \texttt{exp4}, \texttt{exp16}\}$. When it is clear from the context, the subscript $\mathcal{P}$ is dropped from $\Sigma_\mathcal{P}$.

Definition 3. *Let V be a set of variables (assuming $\Sigma \cap V = \emptyset$), the set $T[\Sigma, V]$ of* Σ-terms *over V is inductively defined as follows:*

- $\mathbb{F} \subseteq T[\Sigma, V]$ *and* $V \subseteq T[\Sigma, V]$ *(i.e., every variable/constant is a Σ-term);*
- $\tau \oplus \tau' \in T[\Sigma, V]$ *and* $\tau \otimes \tau' \in T[\Sigma, V]$ *if* $\tau, \tau' \in T[\Sigma, V]$ *(i.e., application of addition and multiplication operators to Σ-terms yield Σ-terms);*
- $f_j(\tau) \in T[\Sigma, V]$ *if* $\tau \in T[\Sigma, V]$ *and* $j \in [1, t]$ *(i.e., application of affine transformations to Σ-terms yield Σ-terms).*

We denote by $T_{\setminus \oplus}(\Sigma, V)$ the set of Σ-terms that do not use the operator $\oplus$.

A Σ-term $\alpha \in T[\Sigma, V]$ is called a *factor* if $\tau \in \mathbb{F} \cup V$ or $\tau = f_i(\tau')$ for some $i \in [1, t]$ such that $\tau' \in T_{\setminus\oplus}(\Sigma, V)$. A *monomial* is a product $\alpha_1 \otimes \cdots \otimes \alpha_k$ of none-zero factors for $k \geq 1$. We denote by $M[\Sigma, V]$ the set of monomials. For instance, consider variables $x, y \in V$ and affine transformations $f_1, f_2 \in \Sigma$. All $f_1(f_2(x)) \otimes f_1(y)$, $f_1(2 \otimes f_2(4 \otimes x))$, $f_1(x \oplus y)$ and $f_1(f_2(x)) \oplus f_1(x)$ are Σ-terms, both $f_1(f_2(x)) \otimes f_1(y)$ and $f_1(2 \otimes f_2(4 \otimes x))$ are monomials, while neither $f_1(x \oplus y)$ nor $f_1(f_2(x)) \oplus f_1(x)$ is a monomial. For the sake of presentation, Σ-terms will be written as terms, and the operator $\otimes$ may be omitted, e.g., $\tau_1 \tau_2$ denotes $\tau_1 \otimes \tau_2$, and τ^2 denotes $\tau \otimes \tau$.

Definition 4. *A polynomial is a sum* $\bigoplus_{i \in [1,t]} m_i$ *of monomials* $m_1 \dots m_t \in M[\Sigma, V]$. *We use* $P[\Sigma, V]$ *to denote the set of polynomials.*

To simplify and normalize polynomials, we impose a total order on monomials and their factors.

Definition 5. *Fix an arbitrary total order* $\geq_s$ *on* $V \uplus \Sigma$.

For two factors α *and* α', *the* factor order $\geq_l$ *is defined such that* $\alpha \geq_l \alpha'$ *if one of the following conditions holds:*

- $\alpha, \alpha' \in \mathbb{F} \cup V$ *and* $\alpha \geq_s \alpha'$;
- $\alpha = f(\tau)$ *and* $\alpha' = f'(\tau')$ *such that* $f \geq_s f'$ *or* $(f = f'$ *and* $\tau \geq_p \tau')$;
- $\alpha = f(\tau)$ *such that* $f \geq_s \alpha'$ *or* $\alpha' = f(\tau)$ *such that* $\alpha \geq_s f$.

Given a monomial $m = \alpha_1 \cdots \alpha_k$, *we write* $\mathsf{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k)$ *for the monomial which includes* $\alpha_1, \cdots, \alpha_k$ *as factors, but sorts them in descending order.*

Given two monomials $m = \alpha_1 \cdots \alpha_k$ *and* $m' = \alpha'_1 \cdots \alpha'_{k'}$, *the* monomial order $\geq_p$ *is defined as the lexicographical order between* $\mathsf{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k)$ *and* $\mathsf{sort}_{\geq_l}(\alpha'_1, \cdots, \alpha'_{k'})$.

Intuitively, the factor order $\geq_l$ follows the given order $\geq_s$ on $V \uplus \Sigma$, where the factor order between two factors with the same affine transformation f is determined by their parameters. We note that if $\mathsf{sort}_{\geq_l}(\alpha'_1, \cdots, \alpha'_{k'})$ is a prefix of $\mathsf{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k)$, we have: $\alpha_1 \cdots \alpha_k \geq_p \alpha'_1 \cdots \alpha'_{k'}$. Furthermore, if $\alpha_1 \cdots \alpha_k \geq_p \alpha'_1 \cdots \alpha'_{k'}$ and $\alpha'_1 \cdots \alpha'_{k'} \geq_p \alpha_1 \cdots \alpha_k$, then $\mathsf{sort}_{\geq_l}(\alpha'_1, \cdots, \alpha'_{k'}) = \mathsf{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k)$. We denote by $\alpha_1 \cdots \alpha_k >_p \alpha'_1 \cdots \alpha'_{k'}$ if $\alpha_1 \cdots \alpha_k \geq_p \alpha'_1 \cdots \alpha'_{k'}$ but $\mathsf{sort}_{\geq_l}(\alpha'_1, \cdots, \alpha'_{k'}) \neq \mathsf{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k)$.

Proposition 1. *The monomial order* $\geq_p$ *is a total order on monomials.*

Definition 6. *Given a program* $\mathcal{P}$, *we define the corresponding term rewriting system (TRS)* $\mathcal{R}$ *as a tuple* $(\Sigma, V, \geq_s, \Delta)$, *where* Σ *is a signature of* $\mathcal{P}$, V *is a set of variables of* $\mathcal{P}$ *(assuming* $\Sigma \cap V = \emptyset$*),* $\geq_s$ *is a total order on* $V \uplus \Sigma$, *and* Δ *is the set of term rewriting rules given below:*

$$R1\frac{(m'_1,\cdots,m'_k)=\texttt{sort}_{\geq_p}(m_1,\cdots,m_k)\neq(m_1,\cdots,m_k)}{m_1\oplus\cdots\oplus m_k\mapsto m'_1\oplus\cdots\oplus m'_k}\quad R3\frac{}{\tau\oplus\tau\mapsto 0}\quad R5\frac{}{0\tau\mapsto 0}$$

$$R2\frac{(\alpha'_1,\cdots,\alpha'_k)=\texttt{sort}_{\geq_l}(\alpha_1,\cdots,\alpha_k)\neq(\alpha_1,\cdots,\alpha_k)}{\alpha_1\cdots\alpha_k\mapsto\alpha'_1\cdots\alpha'_k}\quad R4\frac{}{\tau 0\mapsto 0}\quad R6\frac{}{\tau\oplus 0\mapsto\tau}$$

$$R7\frac{}{0\oplus\tau\mapsto\tau}\quad R8\frac{}{\tau 1\mapsto\tau}\quad R9\frac{}{1\tau\mapsto\tau}\quad R10\frac{}{(\tau_1\oplus\tau_2)\tau\mapsto(\tau_1\tau)\oplus(\tau_2\tau)}$$

$$R11\frac{}{\tau(\tau_1\oplus\tau_2)\mapsto(\tau\tau_1)\oplus(\tau\tau_2)}\quad R12\frac{}{f(\tau_1\oplus\tau_2)\mapsto f(\tau_1)\oplus f(\tau_2)\oplus c}\quad R13\frac{}{f(0)\mapsto c}$$

where $m_1, m'_1, \cdots, m_k, m'_k \in M[\Sigma, V]$, $\alpha_1, \alpha_2, \alpha_3$ *are factors,* $\tau, \tau_1, \tau_2 \in T[\Sigma, V]$ *are terms,* $f \in \Sigma$ *is an affine transformation with affine constant* c.

Intuitively, rules R1 and R2 specify the commutativity of $\oplus$ and $\otimes$, respectively, by which monomials and factors are sorted according to the orders $\geq_p$ and $\geq_l$, respectively. Rule R3 specifies that $\oplus$ is essentially bitwise XOR. Rules R4 and R5 specify that 0 is the multiplicative zero. Rules R6 and R7 (resp. R8 and R9) specify that 0 (resp. 1) is additive (resp. multiplicative) identity. Rules R10 and R11 express the distributivity of $\otimes$ over $\oplus$. Rule R12 expresses the affine property of an affine transformation while rule R13 is an instance of rule R12 via rules R3 and R5.

Given a TRS $\mathcal{R} = (\Sigma, V, \geq_s, \Delta)$ for a given program $\mathcal{P}$, a term $\tau \in T[\Sigma, V]$ can be rewritten to a term τ', denoted by $\tau \Rightarrow \tau'$, if there is a rewriting rule $\tau_1 \mapsto \tau_2$ such that τ' is a term obtained from τ by replacing an occurrence of the sub-term τ_1 with the sub-term τ_2. A term is in a *normal form* if no rewriting rules can be applied. A TRS is *terminating* if all terms can be rewritten to a normal form after finitely many rewriting. We denote by $\tau \Rrightarrow \tau'$ with τ' being the normal form of τ.

We show that any TRS $\mathcal{R}$ associated with a program $\mathcal{P}$ is terminating, and that any term will be rewritten to a normal form that is a polynomial, independent of the way of applying rules.

Lemma 1. *For every normal form* $\tau \in T[\Sigma, V]$ *of the TRS* $\mathcal{R}$*, the term* τ *must be a polynomial* $m_1 \oplus \cdots \oplus m_k$ *such that (1)* $\forall i \in [1, k-1]$, $m_i >_p m_{i+1}$*, and (2) for every monomial* $m_i = \alpha_1 \cdots \alpha_h$ *and* $\forall i \in [1, h-1]$, $\alpha_i \geq_l \alpha_{i+1}$.

Proof. Consider a normal form $\tau \in T[\Sigma, V]$. If τ is not a polynomial, then there must exist some monomial m_i in which the addition operator $\oplus$ is used. This means that either rule R_{10} or R_{11} is applicable to the term τ which contradicts the fact that τ is normal form.

Suppose τ is the polynomial $m_1 \oplus \cdots \oplus m_k$.

- If there exists $i : 1 \leq i < k$ such that $m_i >_p m_{i+1}$ does not hold, then either $m_i = m_{i+1}$ or $m_{i+1} >_p m_i$. If $m_i = m_{i+1}$, then rule R3 is applicable to the term τ. If $m_{i+1} >_p m_i$, then rule R_1 is applicable to the term τ. Thus, for every $1 \leq i < k$, $m_i >_p m_{i+1}$.
- If there exist a monomial $m_i = \alpha_1 \cdots \alpha_h$ and $i : 1 \leq i < h$ such that $\alpha_i \geq_l \alpha_{i+1}$ does not hold, then $\alpha_{i+1} >_l \alpha_i$. This means that rule R2 is applicable to the term τ. Thus, for every monomial $m_i = \alpha_1 \cdots \alpha_h$ and every $i : 1 \leq i < h$, $\alpha_i \geq_l \alpha_{i+1}$. □

Lemma 2. *The TRS* $\mathcal{R} = (\Sigma, V, \geq_s, \Delta)$ *of a given program* $\mathcal{P}$ *is terminating.*

Proof. Consider a term $\tau \in T[\Sigma, V]$. Let $\pi = \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \Rightarrow \cdots \Rightarrow \tau_i \Rightarrow \cdots$ be a reduction of the term τ by applying rewriting rules, i.e., $\tau = \tau_1$. We prove that the reduction π is finite by showing that all the rewriting rules can be applied finitely.

First, since rules R1 and R2 only sort the monomials and factors, respectively, while sorting always terminates using any classic sorting algorithm (e.g., quick sort algorithm), rules R1 and R2 can only be consecutively applied finitely for each term τ_i due to the premises $\texttt{sort}_{\geq_p}(m_1, \cdots, m_k) \neq (m_1, \cdots, m_k)$ and $\texttt{sort}_{\geq_l}(\alpha_1, \cdots, \alpha_k) \neq (\alpha_1, \cdots, \alpha_k)$ in rules R1 and R2, respectively.

Second, rules R10, R11 and R12 can only be applied finitely in the reduction π, as these rules always push the addition operator $\oplus$ toward the root of the syntax tree of the term τ_i when one of them is applied onto a term τ_i, while the other rules either eliminate or reorder the addition operator $\oplus$.

Algorithm 1: Term Normalization

Function `TermNorm`($\mathcal{R}$, τ, λ)**:**
  Rewrite τ by iteratively applying rules R_3–R_{13} until no more update;
  $\tau' \leftarrow \texttt{sort}(\tau)$ by iteratively applying rule R_2;
  $\tau' \leftarrow \texttt{sort}(\tau')$ by iteratively applying rule R_1;
  Rewrite τ' by iteratively applying rules R_3, R_6, R_7 until no more update;
  return τ'

Lastly, rules R3–9 and R13 can only be applied finitely in the reduction π, as these rules reduce the size of the term by 1 when one of them is applied onto a term τ_i while the rules R10–12 that increase the size of the term can only be applied finitely.

Hence, the reduction π is finite indicating that the TRS $\mathcal{R}$ is terminating. □

By Lemmas 1 and 2, any term $\tau \in T[\Sigma, V]$ can be rewritten to a normal form that must be a polynomial.

Theorem 1. *Let* $\mathcal{R} = (\Sigma, V, \geq_s, \Delta)$ *be the TRS of a program* $\mathcal{P}$*. For any term* $\tau \in T[\Sigma, V]$*, a polynomial* $\tau' \in T[\Sigma, V]$ *can be computed such that* $\tau \Rrightarrow \tau'$*.*

Remark 1. Besides the termination of a TRS, confluence is another important property of a TRS, where a TRS is confluent if any given term $\tau \in T[\Sigma, V]$ can be rewritten to two distinct terms τ_1 and τ_2, then the terms τ_1 and τ_2 can be reduced to a common term. While we conjecture that the TRS $\mathcal{R}$ associated with the given program is indeed confluent which may be shown by its local confluence [51], we do not strive to prove its confluence, as it is irrelevant to the problem considered in the current work.

6 Algorithmic Verification

In this section, we first present an algorithm for computing normal forms, then show how to compute the affine constant for an affine transformation, and finally propose an algorithm for solving the verification problem.

6.1 Term Normalization Algorithm

We provide the function `TermNorm` (cf. Algorithm 1) which applies the rewriting rules in a particular order aiming for better efficiency. Fix a TRS $\mathcal{R} = (\Sigma, V, \geq_s, \Delta)$, a term $\tau \in T[\Sigma, V]$ and a mapping λ that provides required affine constants $\lambda(f)$. $\texttt{TermNorm}(\mathcal{R}, \tau, \lambda)$ returns a normal form τ' of τ, i.e., $\tau \Rrightarrow \tau'$.

Algorithm 2: Computing Affine Constants

```
1  Function AffConst(P, R, G):
2      foreach affine transformation f in a topological order of call graph G do
3          if f is only declared in P then
4              λ(f) ← 0;
5          else
6              x ←input of f;
7              ξ(x) ← symbolicExecution(f);
8              τ ← ξ(x)[x ↦ x ⊕ y] ⊕ ξ(x) ⊕ ξ(x)[x ↦ y];
9              while True do
10                 τ ← TermNorm(R, τ, λ);
11                 if τ is some constant c then
12                     λ(f) ← c; break;
13                 else if g is defined in P but has not been inlined in τ then
14                     Inline g in τ; continue;
15                 else if τ does not contain any uninterpreted function then
16                     v1, u1, v2, u2 ←random values from F s.t. v1 ≠ v2 ∨ u1 ≠ u2;
17                     if τ[x ↦ v1, y ↦ u1] ≠ τ[x ↦ v2, y ↦ u2] then
18                         Emit(f is not affine) and Abort;
19                 if SMTSolver(∀x.∀y.τ = c)=SAT then
20                     λ(f) ←extract c from the model; break;
21                 else Emit(f may not be affine) and Abort;
22     return λ;
```

`TermNorm` first applies rules R3–R13 to rewrite the term τ (line 2), resulting in a polynomial which does not have 0 as a factor or monomial (due to rules R4–R7), or 1 as a factor in a monomial unless the monomial itself is 1 (due to rules R_8 and R_9). Next, it recursively sorts all the factors and monomial involved in the polynomial from the innermost sub-terms (lines 3 and 4). Sorting factors and monomials will place the same monomials at adjacent positions. Finally, rules R3 and R6–R7 are further applied to simplify the polynomial (line 5),

where consecutive syntactically equivalent monomials will be rewritten to 0 by rule R3, which may further enable rules R6–R7. Obviously, the final term τ' is a normal form of the input τ, although its size may be exponential in that of τ.

Lemma 3. *`TermNorm`$(\mathcal{R}, \tau, \lambda)$ returns a normal form τ' of τ.* □

6.2 Computing Affine Constants

The function `AffConst` in Algorithm 2 computes the associated affine constant for an affine transformation f. It first sorts all affine transformations in a topological order based on the call graph G (lines 2–21). If f is *only* declared in $\mathcal{P}$, as mentioned previously, we assumed it is linear, thus 0 is assigned to $\lambda(f)$ (line 4). Otherwise, it extracts the input x of f and computes its output $\xi(x)$ via symbolic execution (line 7), where $\xi(x)$ is treated as $f(x)$. We remark that during symbolic execution, we adopt a lazy strategy for inlining invoked affine transformations in f to reduce the size of $\xi(x)$. Thus, $\xi(x)$ may contain affine transformations.

Recall that c is the affine constant of f iff $\forall x, y \in \mathbb{F}. f(x \oplus y) = f(x) \oplus f(y) \oplus c$ holds. Thus, we create the term $\tau = \xi(x)[x \mapsto x \oplus y] \oplus \xi(x) \oplus \xi(x)[x \mapsto y]$ (line 7), where $e[a \mapsto b]$ denotes the substitution of a with b in e. Obviously, the term τ is equivalent to some constant c iff c is the affine constant of f.

The while-loop (lines 9–21) evaluates τ. First, it rewrites τ to a normal form (line 10) by invoking `TermNorm` in Alg.1. If the normal form is some constant c, then c is the affine constant of f. Otherwise, `AffConst` repeatedly inlines each affine transformation g that is defined in P but has not been inlined in τ (lines 13 and 14) and rewrites the term τ to a normal form until either the normal form is some constant c or no affine transformation can be inlined. If the normal form is still not a constant, τ is evaluated using random input values. Clearly, if τ is evaluated to two distinct values (line 18), f is not affine. Otherwise, we check the satisfiability of the constraint $\forall x, y. \tau = c$ via an SMT solver in bitvector theory (line 19), where declared but undefined affine transformations are treated as uninterpreted functions provided with their affine properties. If $\forall x, y. \tau = c$ is satisfiable, we extract the affine constant c from its model (line 20). Otherwise, we emit an error and then abort (line 21), indicating that the affine constant of f cannot be computed. Since the satisfiability problem module bitvector theory is decidable, we can conclude that f is *not* affine if $\forall x. \forall y. \tau = c$ is unsatisfiable and no uninterpreted function is involved in τ.

Lemma 4. *Assume an affine transformation f in $\mathcal{P}$. If `AffConst`$(\mathcal{P}, \mathcal{R}, G)$ in Algorithm 2 returns a mapping λ, then $\lambda(f)$ is the affine constant of f.* □

6.3 Verification Algorithm

The verification problem is solved by the function `Verifier`$(\mathcal{P})$ in Algorithm 3, which checks if $f_{\mathtt{m}} \cong f_{\mathtt{o}}$, for each procedure f defined in $\mathcal{P}$. It first preprocesses the given program $\mathcal{P}$ by inlining all the procedures, unrolling all the loops and

eliminating all the branches (line 2). Then, it computes the corresponding TRS $\mathcal{R}$, call graph G and affine constants as the mapping λ, respectively (line 3). Next, it iteratively checks if $f_{\mathtt{m}} \cong f_{\mathtt{o}}$, for each procedure f defined in $\mathcal{P}$ (lines 4–23).

For each procedure f, it first extracts the inputs $a^1, \cdots, a^m$ of $f_{\mathtt{o}}$ that are scalar variables (line 5) and input encodings $\mathbf{a}^1, \cdots, \mathbf{a}^m$ of $f_{\mathtt{m}}$ that are vectors of variables (line 6). Then, it computes the output $\xi(a^1, \cdots, a^m)$ of $f_{\mathtt{o}}$ via symbolic execution, which yields an expression in terms of $a^1, \cdots, a^m$ and affine transformations (line 7). Similarly, it computes the output $\xi'(\mathbf{a}^1, \cdots, \mathbf{a}^m)$ of $f_{\mathtt{m}}$ via symbolic execution, i.e., a tuple of expressions in terms of the entries of the input encodings $\mathbf{a}^1, \cdots, \mathbf{a}^m$, random variables and affine transformations (line 8).

Recall that $f_{\mathtt{m}} \cong f_{\mathtt{o}}$ iff for all $a^1, \cdots, a^m, r_1, \cdots, r_h \in \mathbb{F}$ and for all $\mathbf{a}^1, \cdots, \mathbf{a}^m \in \mathbb{F}^{d+1}$, the following constraint holds (cf. Definition 1):

$$\left(\bigwedge\nolimits_{i\in[1,m]} a^i = \bigoplus\nolimits_{j\in[0,d]} \mathbf{a}^i_j\right) \rightarrow \left(f_{\mathtt{o}}(a^1, \cdots, a^m) = \bigoplus\nolimits_{i\in[0,d]} f_{\mathtt{m}}(\mathbf{a}^1, \cdots, \mathbf{a}^m_i)\right)$$

where $r_1, \cdots, r_h$ are all the random variables used in $f_{\mathtt{m}}$. Thus, it creates the term $\tau = \xi(a^1, \cdots, a^m)[a^1 \mapsto \bigoplus \mathbf{a}^1, \cdots, a^m \mapsto \bigoplus \mathbf{a}^m] \oplus \bigoplus \xi'(\mathbf{a}^1, \cdots, \mathbf{a}^m)$ (line 9), where $a^i \mapsto \bigoplus \mathbf{a}^i$ is the substitution of a^i with the term $\bigoplus \mathbf{a}^i$ in the expression $\xi(a^1, \cdots, a^m)$. Obviously, τ is equivalent to 0 iff $f_{\mathtt{m}} \cong f_{\mathtt{o}}$.

Algorithm 3: Verification Algorithm

```
1  Function Verifier(P):
2      Inline all the procedures, unroll loops and eliminate branches in P;
3      R ← buildTRS(P); G ← buildCallGraph(P); λ ← AffConst(P, R, G);
4      foreach procedure f defined in P do
5          Let a^1, ··· , a^m be the inputs of f_o;
6          Let a^1, ··· , a^m be the input encodings of f_m;
7          ξ(a^1, ··· , a^m) ← symbolicExecution(f_o);
8          ξ'(a^1, ··· , a^m) ← symbolicExecution(f_m);
9          τ ← ξ(a^1, ··· , a^m)[a^1 ↦ ⊕a^1, ··· , a^m ↦ ⊕a^m] ⊕ ⊕ξ'(a^1, ··· , a^m);
10         while True do
11             τ ← TermNorm(R, τ, λ)
12             if τ is some constant c then
13                 if c = 0 then Emit(f is correct); break;
14                 else Emit(f is incorrect); break;
15             else if g is defined in P but has not been inlined in τ then
16                 Inline g in τ; continue;
17             else if τ does not contain any uninterpreted function then
18                 v^1, ··· , v^m ←random values from F^{d+1};
19                 if τ[a^1 ↦ v^1, ··· , a^m ↦ v^m] ≠ 0 then
20                     Emit(f is incorrect); break;
21             if SMTSolver(τ ≠ 0)=UNSAT then
22                 Emit(f is correct); break;
23             else Emit(f may be incorrect); break;
```

To check if τ is equivalent to 0, similar to computing affine constants in Algorithm 2, the algorithm repeatedly rewrites the term τ to a normal form by invoking `TermNorm` in Algorithm 1 until either the conclusion is drawn or no affine transformation can be inlined (lines 10–23). We declare that f is correct if the normal form is 0 (line 13) and incorrect if it is a non-zero constant (line 14). If the normal form is *not* a constant, we repeatedly inline affine transformation g defined in P which has not been inlined in τ and re-check the term τ.

If there is no definite answer after inlining all the affine transformations, τ is evaluated using random input values. f is *incorrect* if τ is non-zero (line 20). Otherwise, we check the satisfiability of the constraint $\tau \neq 0$ via an SMT solver in bitvector theory (line 21). If $\tau \neq 0$ is unsatisfiable, then f is *correct*. Otherwise we can conclude that f is *incorrect* if no uninterpreted function is involved in τ, but in other cases it is not conclusive.

Theorem 2. *Assume a procedure f in $\mathcal{P}$. If* `Verifier`$(\mathcal{P})$ *emits "f* is correct*", then $f_{\mathtt{m}} \cong f_{\mathtt{o}}$; if* `Verifier`$(\mathcal{P})$ *emits "f* is incorrect*" or "f* may be incorrect*" with no uninterpreted function involved in its final term τ, then $f_{\mathtt{m}} \ncong f_{\mathtt{o}}$.* □

6.4 Implementation Remarks

To implement the algorithms, we use the total order $\geq_s$ on $V \uplus \Sigma$ where all the constants are smaller than the variables, which are in turn smaller than the affine transformations. The order of constants is the standard one on integers, and the order of variables (affine transformations) uses lexicographic order.

In terms of data structure, each term is primarily stored by a directed acyclic graph, allowing us to represent and rewrite common sub-terms in an optimised way. Once a (sub-)term becomes a polynomial during term rewriting, it is stored as a sorted nested list w.r.t. the monomial order $\geq_p$, where each monomial is also stored as a sorted list w.r.t. the factor order $\geq_l$. Moreover, the factor of the form α^k in a monomial is stored by a pair (α, k).

We also adopted two strategies: (i) By Fermat's little theorem [63], $x^{2^n-1} = 1$ for any $x \in \mathbb{GF}(2^n)$. Hence each k in (α, k) can be simplified to $k \mod (2^n - 1)$. (ii) By rule R12, a term $f(\tau_1 \oplus \cdots \oplus \tau_k)$ can be directly rewritten to $f(\tau_1) \oplus \cdots \oplus (\tau_k)$ if k is odd, and $f(\tau_1) \oplus \cdots \oplus f(\tau_k) \oplus c$ if k is even, where c is the affine constant associated with the affine transformation f.

7 Evaluation

We implement our approach as a tool FISCHER for verifying masked programs in LLVM IR, based on the LLVM framework. We first evaluate FISCHER for computing affine constants (i.e., Algorithm 2), correctness verification, and scalability w.r.t. the masking order (i.e., Algorithm 3) on benchmarks using the ISW scheme. To show the generality of our approach, FISCHER is then used to verify benchmarks using glitch-resistant Boolean masking schemes and lattice-based public-key cryptography. All experiments are conducted on a machine

with Linux kernel 5.10, Intel i7 10700 CPU (4.8 GHz, 8 cores, 16 threads) and 40 GB memory. Milliseconds (ms) and seconds (s) are used as the time units in our experiments.

7.1 Evaluation for Computing Affine Constants

To evaluate Algorithm 2, we compare with a pure SMT-based approach which directly checks $\exists c. \forall x, y \in \mathbb{F}. f(x \oplus y) = f(x) \oplus f(y) \oplus c$ using Z3 [47], CVC5 [5] and Boolector [18], by implementing $\oplus$ and $\otimes$ in bit-vector theory, where $\otimes$ is achieved via the Russian peasant method [16]. Technically, SMT solvers only deal with satisfiability, but they usually can eliminate the universal quantifiers in this case, as x, y are over a finite field. In particular, in our experiment, Z3 is configured with default (i.e. `(check-sat)`), simplify (i.e. `(check-sat-using (then simplify smt))`) and bit-blast (i.e. `(check-sat-using (then bit-blast smt))`), denoted by Z3-d, Z3-s and Z3-b, respectively. We focus on the following functions: $\texttt{exp}i(x) = x^i$ for $i \in \{2, 4, 8, 16\}$; $\texttt{rotl}i(x)$ for $i \in \{1, 2, 3, 4\}$ that left rotates x by i bits; $\texttt{af}(x) = \texttt{rotl1}(x) \oplus \texttt{rotl2}(x) \oplus \texttt{rotl3}(x) \oplus \texttt{rotl4}(x) \oplus 99$ used in AES S-Box; $\texttt{L1}(x) = 7x^2 \oplus 14x^4 \oplus 7x^8$, $\texttt{L3}(x) = 7x \oplus 12x^2 \oplus 12x^4 \oplus 9x^8$, $\texttt{L5}(x) = 10x \oplus 9x^2$ and $\texttt{L7}(x) = 4x \oplus 13x^2 \oplus 13x^4 \oplus 14x^8$ used in PRESENT S-Box over $\mathbb{GF}(16) = \mathbb{GF}(2)[X]/(X^4 + X + 1)$ [14,19]; $\texttt{f1}(x) = x^3$, $\texttt{f2}(x) = x^2 \oplus x \oplus 1$, $\texttt{f3}(x) = x \oplus x^5$ and $\texttt{f4}(x) = \texttt{af}(\texttt{exp2}(x))$ over $\mathbb{GF}(2^8)$.

Table 1. Results of computing affine constants, where † means Algorithm 2 needs SMT solving, ‡ means affineness is disproved via testing, ✗ means nonaffineness, and Algorithm 2+B means Algorithm 2+Boolector.

Tool	exp2	exp4	exp8	exp16	rotl1	rotl2	rotl3	rotl4	af	L1	L3	L5	L7	f1	f2	f3	f4
Algorithm 2+Z3-d	3 ms	3 ms	3 ms	3 ms	18 ms†	18 ms†	18 ms†	18 ms†	21 ms†	3 ms	3 ms	3 ms	3 ms	3 ms‡	3 ms	3 ms‡	21 ms†
Algorithm 2+Z3-b	3 ms	3 ms	3 ms	3 ms	15 ms†	16 ms†	15 ms†	15 ms†	20 ms†	3 ms	3 ms	3 ms	3 ms	3 ms‡	3 ms	3 ms‡	20 ms†
Algorithm 2+B	3 ms	3 ms	3 ms	3 ms	8 ms†	8 ms†	8 ms†	8 ms†	13 ms†	3 ms	3 ms	3 ms	3 ms	3 ms‡	3 ms	3 ms‡	14 ms†
Z3-d	181 ms	333 ms	316 ms	521 ms	14 ms	14 ms	14 ms	14 ms	16 ms	113 ms	213 ms	73 ms	194 ms	33 ms	249 ms	38 ms	7.5s
Z3-s	180 ms	373 ms	452 ms	528 ms	12 ms	12 ms	12 ms	12 ms	15 ms	158 ms	202 ms	194 ms	213 ms	28 ms	252 ms	35 ms	7.6s
Z3-b	15 ms	16 ms	18 ms	20 ms	12 ms	12 ms	12 ms	12 ms	79 ms	45 ms	42 ms	21 ms	82 ms	17 ms	22 ms	24 ms	60 ms
Boolector	15 ms	18 ms	12 ms	17 ms	5 ms	5 ms	6 ms	5 ms	71 ms	25 ms	34 ms	27 ms	78 ms	14 ms	15 ms	17 ms	67 ms
CVC5	8.4 s	20.3 s	44.4 s	18.6 s	5 ms	5 ms	5 ms	5 ms	113 ms	158.4 s	263.4 s	43.7 s	214.9 s	92 ms	10.3 s	2.3 s	10.4 s
Result	0	0	0	0	0	0	0	0	99	0	0	0	0	✗	1	✗	99

The results are reported in Table 1, where the 2nd–8th rows show the execution time and the last row shows the affine constants if they exist otherwise ✗. We observe that Algorithm 2 significantly outperforms the SMT-based approach on most cases for all the SMT solvers, except for $\texttt{rotl}i$ and af (It is not surprising, as they use operations rather than $\oplus$ and $\otimes$, thus SMT solving is required). The term rewriting system is often able to compute affine constants *solely* (e.g., $\texttt{exp}i$ and $\texttt{L}i$), and SMT solving is required *only* for computing the affine constants of $\texttt{rotl}i$. By comparing the results of Algorithm 2+Z3-b vs. Z3-b and Algorithm 2+B vs. Boolector on af, we observe that term rewriting is essential as checking normal form—instead of the original constraint—reduces the cost of SMT solving.

7.2 Evaluation for Correctness Verification

To evaluate Algorithm 3, we compare it with a pure SMT-based approach with SMT solvers Z3, CVC5 and Boolector. We also consider several promising general-purpose software verifiers SMACK (with Boogie and Corral engines), SeaHorn, CPAChecker and Symbiotic, and one cryptography-specific verifier CryptoLine (with SMT and CAS solvers), where the verification problem is expressed using assume and assert statements. Those verifiers are configured in two ways: (1) recommended ones in the manual/paper or used in the competition, and (2) by trials of different configurations and selecting the optimal one. Specifically:

- CryptoLine (commit `7e237a9`). Both solvers SMT and CAS are used;
- SMACK v2.8.0. integer-encoding: bit-vector, verifier: corral/boogie (both used), solver: Z3/CVC4 (Z3 used), static-unroll: on, unroll: 99;
- SEAHORN v0.1.0 RC3 (commit `e712712`). pipeline: bpf, arch: m64, inline: on, track: mem, bmc: none/mono/path (mono used), crab: on/off (off used);
- CPAChecker v2.1.1. default.properties with cbmc: on/off (on used);
- Symbiotic v8.0.0. officially-provided SV-COMP configuration with exit-on-error: on.

The benchmark comprises five different masked programs `sec_mult` for finite-field multiplication over $\mathbb{GF}(2^8)$ by varying masking order $d = 0, 1, 2, 3$, where the $d = 0$ means the program is unmasked. We note that `sec_mult` in [8] is only available for masking order $d \geq 2$.

Table 2. Results on various `sec_mult`, where T.O. means time out (20 min), N/A means that UNKNOWN result, and ↯ means that verification result is *incorrect*.

Order	Ref.	Algorithm 3	Z3			Boolector	CVC5	CryptoLine		SMACK		SeaHorn	CPAChecker	Symbiotic
d			default	simplify	bit-blast			SMT	CAS	Boogie	Corral			
0	[58]	17 ms	29 ms	27 ms	42 ms	25 ms	29 ms	39 ms	N/A	29 s	66 s	132 ms	T.O	870 s
	[11]	20 ms	31 ms ms	31 ms	45 ms	28 ms	33 ms	35 ms	N/A	46 s	144 s	128 ms	T.O	899 s
	[34]	21 ms	33 ms	31 ms	46 ms	29 ms	33 ms	32 ms	N/A	23 s	43 s	127 ms	T.O	872 s
	[21]	18 ms	30 ms	28 ms	25 ms	26 ms	31 ms	32 ms	N/A	17 s	56 s	130 ms	T.O	876 s
1	[58]	18 ms	298 ms	299 ms	391 s	3.8 s	T.O	469 ms	N/A	T.O	T.O	13 s	T.O	T.O
	[11]	20 ms	299 ms	299 ms	1049 s	1.91049	T.O	582 ms	N/A	T.O	T.O	13 s	T.O	T.O
	[34]	24 ms	295 ms	295 ms	1199 s	1.8 s	T.O	951 ms	N/A	T.O	T.O	14 s	T.O	T.O
	[21]	20 ms	1180 s	921 s	T.O	7.7 s	T.O	21 s	N/A	T.O	T.O	T.O	T.O	T.O.
2	[58]	20 ms	4.1 s	4.2 s	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[11]	22 ms	4.2 s	4.4 s	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[8]	30 ms	4.2 s	4.1 s	T.O	T.O	T.O	T.O	N/A	T.O	26 s↯	T.O	T.O	T.O
	[34]	29 ms	4.2 s	4.2 s	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[21]	22 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O.
3	[58]	21 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[11]	26 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[8]	27 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	1059 s↯	T.O	T.O	T.O
	[34]	29 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O
	[21]	24 ms	T.O	T.O	T.O	T.O	T.O	T.O	N/A	T.O	T.O	T.O	T.O	T.O

The results are shown in Table 2. We can observe that FISCHER is significantly more efficient than the others, and is able to prove all the cases using

our term rewriting system *solely* (i.e., without random testing or SMT solving). With the increase of masking order d, almost all the other tools failed. Both CryptoLine (with the CAS solver) and CPAChecker fail to verify any of the cases due to the non-linear operations involved in `sec_mult`. SMACK with Corral engine produces two false positives (marked by ♮ in Table 2). These results suggest that dedicated verification approaches are required for proving the correctness of masked programs.

7.3 Scalability of FISCHER

To evaluate the scalability of FISCHER, we verify different versions of `sec_mult` and masked procedures `sec_aes_sbox` (resp. `sec_present_sbox`) of S-Boxes used in AES [58] (resp. PRESENT [19]) with varying masking order d. Since it is known that `refresh_masks` in [58] is vulnerable when $d \geq 4$ [24], a fixed version `RefreshM` [7] is used in all the S-Boxes (except that when `sec_mult` is taken from [8] its own version is used). We note that `sec_present_sbox` uses the affine transformations `L1`, `L3`, `L5`, `L7`, `exp2` and `exp4`, while `sec_aes_sbox` uses the affine transformations `af`, `exp2`, `exp4` and `exp16`.

The results are reported in Table 3. All those benchmarks are proved using our term rewriting system solely except for the three incorrect ones marked by ♮. FISCHER scales up to masking order of 100 or even 200 for `sec_mult`, which is remarkable. FISCHER also scales up to masking order of 30 or even 40 for `sec_present_sbox`. However, it is less scalable on `sec_aes_sbox`, as it computes the multiplicative inverse x^{254} on shares, and the size of the term encoding the equivalence problem explodes with the increase of the masking order. Furthermore, to better demonstrate the effectiveness of our term writing system in dealing with complicated procedures, we first use Algorithm 2 to derive affine constants on `sec_aes_sbox` with ISW [58] and then directly apply SMT solvers to solve the correctness constraints obtained at Line 9 of Algorithm 3. It takes about 1 s to obtain the result on the first-order masking, while fails to obtain the result within 20 min on the second-order masking.

Table 3. Results on `sec_mult` and S-Boxes, where T.O. means time out (20 min), and ♮ means that the program is *incorrect*.

Ref.	d																
	`sec_mult`						`sec_present_sbox`							`sec_aes_sbox`			
	5	10	20	50	100	200	1	2	5	10	20	30	40	1	2	4	5
ISW [58]	23 ms	33 ms	84 ms	1.0s	15s	545s	44 ms	51 ms	93 ms	535 ms	14s	118s	T.O.	87 ms	234 ms	25s	160s
ISW [11]	26 ms	44 ms	100 ms	712 ms	7.3s	212s	54 ms	63 ms	110 ms	673 ms	17s	163s	T.O.	108 ms	265 ms	23s	142s
ISW [8]	36 ms♮	49 ms	109 ms	601 ms	3.2s	18s	–	86 ms	142 ms♮	237 ms	841 ms	2.4s	5.3s	–	559 ms	9.7s	142s♮
ISW [34]	34 ms	50 ms	98 ms	518 ms	3.1s	19s	67 ms	91 ms	137 ms	700 ms	20s	173s	T.O.	140 ms	571 ms	63s	T.O.
ISW [21]	30 ms	109 ms	224 ms	5.0s	152s	T.O.	51 ms	61 ms	113 ms	354 ms	2.4s	9.7s	29s	133 ms	269 ms	13s	68s

Table 4. Results on `sec_mult` and S-Boxes for HPC, DOM and CMS.

Ref.	d															
	`sec_mult`						`sec_present_sbox`					`sec_aes_sbox`				
	0	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
HPC1 [20]	28 ms	30 ms	32 ms	35 ms	39 ms	42 ms	63 ms	72 ms	84 ms	98 ms	117 ms	104 ms	254 ms	1.8s	13s	67s
HPC2 [20]	23 ms	25 ms	26 ms	28 ms	31 ms	33 ms	57 ms	66 ms	75 ms	92 ms	110 ms	92 ms	244 ms	1.9s	13s	65s
DOM [35]	24 ms	24 ms	25 ms	26 ms	28 ms	29 ms	52 ms	60 ms	67 ms	77 ms	90 ms	80 ms	223 ms	1.8s	12s	66s
CMS [57]	–	–	24 ms	–	–	–	–	53 ms	–	–	–	–	211 ms	–	–	–

A highlight of our findings is that FISCHER reports that `sec_mult` from [8] and the S-boxes based on this version are incorrect when $d = 5$. After a careful analysis, we found that indeed it is incorrect for any $d \equiv 1 \mod 4$ (i.e., 5, 9, 13, etc.). This is because [8] parallelizes the multiplication over the entire encodings (i.e., tuples of shares) while the parallelized computation depends on the value of $d \mod 4$. When the reminder is 1, the error occurs.

7.4 Evaluation for More Boolean Masking Schemes

To demonstrate the applicability of FISCHER on a wider range of Boolean masking schemes, we further consider glitch-resistant Boolean masking schemes: HPC1, HPC2 [20], DOM [35] and CMS [57]. We implement the finite-field multiplication `sec_mult` using those masking schemes, as well as masked versions of AES S-box and PRESENT S-box. We note that our implementation of DOM `sec_mult` is derived from [20], and we only implement the 2nd-order CMS `sec_mult` due to the difficulty of implementation. All other experimental settings are the same as in Sect. 7.3.

The results are shown in Table 4. Our term rewriting system *solely* is able to efficiently prove the correctness of finite-field multiplication `sec_mult`, masked versions of AES S-box and PRESENT S-box using the glitch-resistant Boolean masking schemes HPC1, HPC2, DOM and CMS. The verification cost of those benchmarks is similar to that of benchmarks using the ISW scheme, demonstrating the applicability of FISCHER for various Boolean masking schemes.

Table 5. Results on `sec_add`, `sec_add_modp` and `sec_a2b` [17], where T.O. means time out (20 min).

d	k																		
	`sec_add`							`sec_add_modp`					`sec_a2b`						
	2	3	4	6	8	12	16	2	3	4	6	8	12	2	3	4	6	8	12	16
1	34 ms	38 ms	42 ms	51 ms	61 ms	83 ms	109 ms	97 ms	248 ms	805 ms	7.5s	44s	623s	41 ms	48 ms	55 ms	70 ms	87 ms	121 ms	156 ms
2	35 ms	40 ms	45 ms	55 ms	65 ms	91 ms	124 ms	111 ms	331 ms	1.1s	11s	67s	936s	58 ms	74 ms	93 ms	134 ms	199 ms	523 ms	1.5s
3	36 ms	42 ms	47 ms	58 ms	71 ms	100 ms	139 ms	127 ms	417 ms	1.5s	15s	89s	T.O.	73 ms	93 ms	118 ms	182 ms	293 ms	927 ms	3.0s
4	38 ms	44 ms	50 ms	62 ms	76 ms	109 ms	155 ms	144 ms	506 ms	1.9s	18s	112s	T.O.	93 ms	130 ms	190 ms	676 ms	3.3s	49s	366s
5	39 ms	45 ms	51 ms	66 ms	81 ms	118 ms	168 ms	160 ms	586 ms	2.2s	22s	136s	T.O.	109 ms	159 ms	256 ms	1.1s	6.5s	100s	746s

7.5 Evaluation for Arithmetic/Boolean Masking Conversions

To demonstrate a wider applicability of FISCHER other than masked implementations of symmetric cryptography, we further evaluate FISCHER on three key non-linear building blocks for bitsliced, masked implementations of lattice-based post-quantum key encapsulation mechanisms (KEMs [17]). Note that KEMs are a class of encryption techniques designed to secure symmetric cryptographic key material for transmission using asymmetric (public-key) cryptography. We implement the Boolean masked addition modulo 2^k (`sec_add`), Boolean masked addition modulo p (`sec_add_modp`) and the arithmetic-to-Boolean masking conversion modulo 2^k (`sec_a2b`) for various bit-width k and masking order d, where p is the largest prime number less than 2^k. Note that some bitwise operations (e.g., circular shift) are expressed by affine transformations, and the modulo addition is implemented by the simulation algorithm [17] in our implementations.

The results are reported in Table 5. FISCHER is able to efficiently prove the correctness of these functions with various masking orders (d) and bit-width (k), using the term rewriting system *solely*. With the increase of the bit-width k (resp. masking order d), the verification cost increases more quickly for `sec_add_modp` (resp. `sec_a2b`) than for `sec_add`. This is because `sec_add_modp` with bit-width k invokes `sec_add` three times, two of which have the bit-width $k+1$, and the number of calls to `sec_add` in `sec_a2b` increases with the masking order d though using the same bit-width as `sec_a2b`. These results demonstrate the applicability of FISCHER for asymmetric cryptography.

8 Conclusion

We have proposed a term rewriting based approach to proving functional equivalence between masked cryptographic programs and their original unmasked algorithms over $\mathbb{GF}(2^n)$. Based on this approach, we have developed a tool FISCHER and carried out extensive experiments on various benchmarks. Our evaluation confirms the effectiveness, efficiency and applicability of our approach.

For future work, it would be interesting to further investigate the theoretical properties of the term rewriting system. Moreover, we believe the term rewriting approach extended with more operations may have a greater potential in verifying more general cryptographic programs, e.g., those from the standard software library such as OpenSSL.

References

1. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innov. Syst. Softw. Eng. **9**(2), 59–77 (2013)
2. Ahman, D., et al.: Dijkstra monads for free. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 515–529 (2017)

3. Almeida, J.B., et al.: Jasmin: high-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1807–1823 (2017)
4. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2), 1–31 (2015)
5. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 13243, pp. 415–442 (2022). v1.0.0 is used
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
7. Barthe, G., et al.: Strong non-interference and type-directed higher-order masking. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 116–129 (2016)
8. Barthe, G., Dupressoir, F., Faust, S., Grégoire, B., Standaert, F.-X., Strub, P.-Y.: Parallel implementations of masking schemes and the bounded moment leakage model. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 535–566. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_19
9. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.-Y.: EasyCrypt: a tutorial. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) FOSAD 2012-2013. LNCS, vol. 8604, pp. 146–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10082-1_6
10. Beierle, C., et al.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9815, pp. 123–153. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53008-5_5
11. Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., Thillard, A., Vergnaud, D.: Randomness complexity of private circuits for multiplication. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 616–648. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_22
12. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
13. Bloem, R., Gross, H., Iusupov, R., Könighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 321–353. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_11
14. Bogdanov, A., et al.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_31
15. Bond, B., et al.: Vale: verifying high-performance cryptographic assembly code. In: 26th USENIX security symposium, pp. 917–934 (2017)
16. Bowden, J.: The Russian peasant method of multiplication. Math. Teach. **5**(1), 4–8 (1912)
17. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/Boolean masking conversions for fun and profit: with application to lattice-based KEMs. IACR Trans. Cryptograph. Hardw. Embed. Syst., 553–588 (2022)

18. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_16
19. Carlet, C., Goubin, L., Prouff, E., Quisquater, M., Rivain, M.: Higher-order masking schemes for s-boxes. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 366–384. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34047-5_21
20. Cassiers, G., Grégoire, B., Levi, I., Standaert, F.X.: Hardware private circuits: from trivial composition to full verification. IEEE Trans. Comput. **70**(10), 1677–1690 (2020)
21. Cassiers, G., Standaert, F.X.: Trivially and efficiently composing masked gadgets with probe isolating non-interference. IEEE Trans. Inf. Forensics Secur. **15**, 2542–2555 (2020)
22. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: beyond symbolic execution. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12652, pp. 453–457. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_31
23. Chen, Y.F., et al.: Verifying curve25519 software. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 299–309 (2014)
24. Coron, J.-S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 410–424. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43933-3_21
25. Daemen, J., Rijmen, V.: AES proposal: Rijndael (1999)
26. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. **24**(2), 11 (2014)
27. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy, pp. 1202–1219 (2019)
28. Fu, Y., Liu, J., Shi, X., Tsai, M., Wang, B., Yang, B.: Signed cryptographic program verification with typed cryptoline. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1591–1606 (2019)
29. Gao, P., Xie, H., Song, F., Chen, T.: A hybrid approach to formal verification of higher-order masked arithmetic programs. ACM Trans. Softw. Eng. Methodol. **30**(3), 1–42 (2021)
30. Gao, P., Xie, H., Sun, P., Zhang, J., Song, F., Chen, T.: Formal verification of masking countermeasures for arithmetic programs. IEEE Trans. Software Eng. **48**(3), 973–1000 (2022)
31. Gao, P., Xie, H., Zhang, J., Song, F., Chen, T.: Quantitative verification of masked arithmetic programs against side-channel attacks. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 155–173. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_9
32. Gao, P., Zhang, J., Song, F., Wang, C.: Verifying and quantifying side-channel resistance of masked software implementations. ACM Trans. Softw. Eng. Methodol. **28**(3), 1–32 (2019)
33. Goubin, L., Patarin, J.: DES and differential power analysis the "Duplication" method. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48059-5_15

34. Gross, H., Mangard, S.: Reconciling $d + 1$ masking in hardware and software. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 115–136. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_6
35. Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive (2016)
36. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
37. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_27
38. Kaufmann, D., Biere, A.: AMulet 2.0 for verifying multiplier circuits. In: TACAS 2021. LNCS, vol. 12652, pp. 357–364. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_19
39. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: Proceedings of the 2019 Formal Methods in Computer Aided Design, pp. 28–36 (2019)
40. Kim, H.S., Hong, S., Lim, J.: A fast and provably secure higher-order masking of AES S-Box. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 95–107. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_7
41. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
42. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
43. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization, pp. 75–86 (2004)
44. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
45. Liu, J., Shi, X., Tsai, M., Wang, B., Yang, B.: Verifying arithmetic in cryptographic C programs. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 552–564 (2019)
46. Luo, C., Fei, Y., Kaeli, D.R.: Effective simple-power analysis attacks of elliptic curve cryptography on embedded systems. In: Proceedings of the International Conference on Computer-Aided Design, p. 115 (2018)
47. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
48. Myreen, M.O., Curello, G.: Proof pearl: a verified Bignum implementation in x86-64 machine code. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 66–81. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_5
49. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 568–582. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_44

50. National Institute of Standards and Technology: Data encryption standard (DES). FIPS Publication, pp. 46–3, October 1999
51. Newman, M.H.A.: On theories with a combinatorial definition of equivalence. Annals Math., 223–243 (1942)
52. Örs, S.B., Oswald, E., Preneel, B.: Power-analysis attacks on an FPGA – first experimental results. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 35–50. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45238-6_4
53. Polyakov, A., Tsai, M., Wang, B., Yang, B.: Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In: Proceedings of the 29th International Conference on Concurrency Theory, pp. 1–16 (2018)
54. Prouff, E., Rivain, M., Bevan, R.: Statistical analysis of second order differential power analysis. IEEE Trans. Comput. **58**(6), 799–811 (2009)
55. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
56. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEM schemes. IACR Cryptol. ePrint Arch. **2019**, 948 (2019)
57. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 764–783. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_37
58. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_28
59. Schamberger, T., Renner, J., Sigl, G., Wachter-Zeh, A.: A power side-channel attack on the CCA2-secure HQC KEM. In: Liardet, P.-Y., Mentens, N. (eds.) CARDIS 2020. LNCS, vol. 12609, pp. 119–134. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68487-7_8
60. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
61. Tomb, A.: Automated verification of real-world cryptographic implementations. IEEE Secur. Priv. **14**(6), 26–33 (2016)
62. Tsai, M.H., Wang, B.Y., Yang, B.Y.: Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1973–1987 (2017)
63. Vinogradov, I.M.: Elements of Number Theory. Courier Dover Publications, New York (2016)
64. Zhang, J., Gao, P., Song, F., Wang, C.: SCInfer: refinement-based verification of software countermeasures against side-channel attacks. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 157–177. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_12
65. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1789–1806 (2017)