PoS4MPC: Automated Security Policy Synthesis for Secure Multi-Party Computation*

Yuxin Fan[1], Fu Song[1,2(✉)], Taolue Chen[3], Liangfeng Zhang[1], and Wanwei Liu[4,5]

[1] School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China
[2] Shanghai Engineering Research Center of Intelligent Vision and Imaging, Shanghai 201210, China
[3] Department of Computer Science, Birkbeck, University of London, WC1E 7HX, UK
[4] College of Computer Science, National University of Defense Technology, Changsha 410073, China
[5] State Key Laboratory for High Performance Computing, Changsha 410073, China

Abstract. Secure multi-party computation (MPC) is a promising technique for privacy-persevering applications. A number of MPC frameworks have been proposed to reduce the burden of designing customized protocols, allowing non-experts to quickly develop and deploy MPC applications. To improve performance, recent MPC frameworks allow users to declare variables secret only for these which are to be protected. However, in practice, it is usually highly non-trivial for non-experts to specify secret variables: declaring too many degrades the performance while declaring too less compromises privacy. To address this problem, in this work we propose an automated security policy synthesis approach to declare as few secret variables as possible but without compromising security. Our approach is a synergistic integration of type inference and symbolic reasoning. The former is able to quickly infer a sound—but sometimes conservative—security policy, whereas the latter allows to identify secret variables in a security policy that can be declassified in a precise manner. Moreover, the results from symbolic reasoning are fed back to type inference to refine the security types even further. We implement our approach in a new tool **PoS4MPC**. Experimental results on five typical MPC applications confirm the efficacy of our approach.

1 Introduction

Secure multi-party computation (MPC) is a powerful cryptographic paradigm, allowing mutually distrusting parties to collaboratively compute a public function over their private data without a trusted third party and revealing nothing

* This work is supported by the National Natural Science Foundation of China (NSFC) under Grants No. 62072309, No. 61872340 and No. 61872371, the Open Fund from the State Key Laboratory of High Performance Computing of China (HPCL) (202001-07), an overseas grant from the State Key Laboratory of Novel Software Technology, Nanjing University, and Birkbeck BEI School Project (EFFECT).

beyond the result of the computation and their own private data [43,14]. MPC has potential for broader uses in practical applications, e.g., truthful auctions, avoiding satellite collisions [22], private machine learning [41], and data analysis [35]. However, practical deployment of MPC has been limited due to its computational and communication complexity.

To foster applications of MPC, a number of general-purpose MPC frameworks have been proposed, e.g., [9,34,29,44,37,24]. These frameworks provide high-level languages for specifying MPC applications as well as compilers for translating them into executable implementations, thus drastically reduce the burden of designing customized protocols and allow non-experts to quickly develop and deploy MPC applications. To improve performance, many MPC frameworks provide features to declare secret variables so that only these variables are to be protected. However, such frameworks usually do not verify rigorously whether there is information leakage, or, on some occasions, provide only light-weighted checking (via, e.g., information-flow analysis). Even though some frameworks are equipped with formal security guarantees, it is challenging for non-experts to develop an MPC program that simultaneously achieves good performance and formal security guarantees [28,3]. A typical case for an user is to declare all variables secret while ideally one would declare as few secret variables as possible to achieve a good performance without compromising security.

In this work, we propose an automated security policy synthesis approach for MPC. We first formalize the leakage of an MPC application in the ideal-world as a set of private inputs and define the notion of security policy, which assigns each variable a security level. This can bridge the language-level and protocol-level leakages, hence our approach is independent of the specific MPC protocols being used. Based on the leakage characterization, we provide a type system to infer security policies by tracking both control- and data-flow of information from private inputs. While a security policy inferred from the type system formally guarantees that the MPC application will not leak more information than the result of the computation and participants' own private data, it may be too conservative. For instance, some variables could be declassified without compromising security but with improved performance. Therefore, we propose a symbolic reasoning approach to identify secret variables in security policies that can be declassified without compromising security. We also feed back the results from the symbolic reasoning to type inference to refine the security type further.

We implement our approach in a new tool **PoS4MPC** (**Po**licy **S**ynthesis for **MPC**) based on the LLVM Compiler [1] and the KLEE symbolic execution engine [10]. Experimental results on five typical MPC applications show that our approach can generate less restrictive security policies than using the type system solely. We also deploy the generated security policies in two MPC frameworks Obliv-C [44] and MPyC [37]. The results show that, for instance, the security policies generated by our approach can reduce the execution time by 31%–1.56×10^5%, the circuit size by 38%–3.61×10^5%, and the communication traffic by 39%–4.17×10^5% in Obliv-C.

To summarize, our main technical contributions are as follows.

```
int demo(int a, int b, int c){
    int r = 1;  int max = a;
    bool c1 = max < b;
    if (c1){ max = b; r = 2;  }
    bool c2 = max < c;
    if (c2){ r = 3; }
    return r; }
```

Fig. 1: The richest one of three millionaires

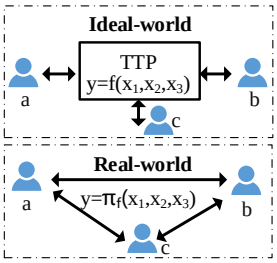

Fig. 2: Ideal-world vs. real-world

– A formalization of information leakage for MPC applications and the notion of security policy to bridge the language-level and protocol-level leakages;
– An automated security policy synthesis approach that is able to generate less restrictive security policies;
– An implementation of our approach for a real-world language and an evaluation on challenging benchmarks from the literature.

Outline. Section 2 presents the motivation of this work and overview of our approach. Section 3 gives the background of MPC. Section 4 introduces a simple language on which we formalize the leakage of MPC applications. We propose a type system for inferring security policies in Section 5 and a symbolic reasoning approach for declassification in Section 6. Implementation details and experimental results are given in Section 7. Finally, we discuss related work in Section 8 and conclude this paper in Section 9.

Missing proofs can be found in the full version of this paper [15].

2 Motivation

Fig. 1 shows a motivating example that computes the richest among three millionaires. To preserve the privacy, the millionaires can privately send their inputs to a trusted third party (TTP) as shown in Fig. 2 (ideal-world). This reveals the richest millionaire with the least leakage of information. Table 1 shows the leakage for each result $\mathtt{r} = 1, 2, 3$, as well as the leakage if the secret branching variables `c1` and `c2` are declassified (i.e., from secret to public).

Table 1: Leakage from each result and declassified secret branching variables

Result	Leakage of Result	Leakage of c1	Leakage of c2
$\mathtt{r} = 1$	$\mathtt{a} \geq \mathtt{b} \wedge \mathtt{a} \geq \mathtt{c}$	$\mathtt{a} \geq \mathtt{b}$	$\mathtt{a} \geq \mathtt{c}$
$\mathtt{r} = 2$	$\mathtt{a} < \mathtt{b} \wedge \mathtt{b} \geq \mathtt{c}$	$\mathtt{a} < \mathtt{b}$	$\mathtt{b} \geq \mathtt{c}$
$\mathtt{r} = 3$	$\mathtt{c} > \max(\mathtt{a}, \mathtt{b})$	$\mathtt{a} \geq \mathtt{b} \vee \mathtt{a} < \mathtt{b}$	$\mathtt{c} > \max(\mathtt{a}, \mathtt{b})$

To achieve the same functionality without TTP, secure multi-party computation (MPC) was proposed [43,14]. One can implement the computation using

an MPC protocol π where all the parties collaboratively compute the result over their private inputs via network communications (shown in Fig. 2 (real-world)).

To facilitate applications of MPC, various MPC frameworks, e.g., Obliv-C [44], MP-SPDZ [24] and MPyC [37], have been proposed, which provide high-level languages for specifying MPC applications, as well as compilers for translating them into executable implementations. To improve performance, these frameworks often allow users to declare secret variables so that only the values of secret variables are to be protected. However, in practice, it is usually quite challenging for non-experts to specify secret variables properly: declaring too many secret variables would degrade the performance, whereas declaring too less secret variables risks compromising security and privacy.

In this work, we propose an automated synthesis approach, aiming to declare as few secret variables as possible but without compromising security. To capture privacy, we formalize the leakage of MPC applications in the ideal-world as a set of private inputs. For instance, the leakage of the result $\mathtt{r} = 1$ in the motivating example is the set of inputs such that $\mathtt{a} \geq \mathtt{b} \wedge \mathtt{a} \geq \mathtt{c}$. We introduce the notion of security policy, which assigns each variable a security level, to bridge the language-level and protocol-level leakages, so that our approach is independent of specific MPC protocols being used. The language-level leakage of a security policy is characterized by a set of private inputs with respect to not only the result but also the values of public variables in the intermediate computations.

Based on the leakage characterization, we propose a type system to automatically infer security policies, inspired by the work of proving noninterference of programs [40]. Our type system tracks both control-flow and data-flow of information from the private inputs, and infers a security policy. For instance, all the variables in the motivating example are inferred as secret.

Although a security policy inferred by the type system formally guarantees that the MPC application will not leak more information than that in the ideal-world, it may be too conservative. For instance, declassifying the variable `c2` in the example would not compromise security. As shown in Table 1, the leakage caused by declassifying `c2` can be deduced from the leakage of the result. In contrast, we cannot declassify `c1`, as neither $\mathtt{a} \geq \mathtt{b}$ nor $\mathtt{a} < \mathtt{b}$ can be deduced from the leakage $\mathtt{c} > \max(\mathtt{a}, \mathtt{b})$. Once `c1` is declassified, the adversary would learn if $\mathtt{a} \geq \mathtt{b}$ or $\mathtt{a} < \mathtt{b}$. This problem is akin to downgrading and declassification of high security levels in information-flow analysis [27], and could be solved via self-composition [39,42] that often require users to write annotations for procedure contracts and loop invariants. In this work, for the sake of efficiency and usability for non-experts, we propose an alternative approach based on symbolic execution. We leverage symbolic execution to finitely represent a potentially infinite set of concrete executions, and propose an automated approach to infer if a secret variable can be declassified by reasoning about pairs of symbolic executions. For instance, in Example 1, our approach is able to identify that `c2` can be declassified without compromising security. In general, the experimental results show that our approach is effective and the generated security policies can significantly improve the performance of MPC applications.

3 Secure MPC

Fix a set of variables $\mathcal{X}$ over a domain $\mathcal{D}$. We write $\overline{\mathbf{x}}_n \in \mathcal{X}^n$ and $\overline{\mathbf{v}}_n \in \mathcal{D}^n$ for tuples $(x_1, \cdots, x_n)$ and $(v_1, \cdots, v_n)$ respectively. (The subscript n may be dropped when it is clear from the context.)

MPC in the ideal-world. An n-party MPC application $f : \mathcal{D}^n \to \mathcal{D}$ is to confidentially compute a given function $f(\overline{\mathbf{x}})$, where each party $\mathsf{P_i}$ for $1 \leq i \leq n$ sends her private input $v_i \in \mathcal{D}$ to a TTP T which computes and returns the result $f(\overline{\mathbf{v}})$ to all the parties. In the ideal world, an adversary that controls any of the n parties learns no more than the output $f(\overline{\mathbf{v}})$ and the private inputs of the corrupted (dishonest) parties.

We characterize the leakage of an MPC application $f(\overline{\mathbf{x}})$ by a set of private inputs. Hereafter, we assume, w.l.o.g., the first k parties (i.e., $\mathsf{P_1}, \cdots, \mathsf{P_k}$) are corrupted by the adversary for some $k \geq 1$. For a given output $v \in \mathcal{D}$, let $\simeq_v^f \subseteq \mathcal{D}^n$ be the set $\{\overline{\mathbf{v}} \in \mathcal{D}^n \mid f(\overline{\mathbf{v}}) = v\}$. Intuitively, $\simeq_v^f$ is the set of the private inputs $\overline{\mathbf{v}} \in \mathcal{D}^n$ under which f is evaluated to v. From the result v, the adversary is able to learn the set $\simeq_v^f$, but cannot tell which one from $\simeq_v^f$ given v. We refer to $\simeq_v^f$ as the indistinguishable space of the private inputs w.r.t. the result v. The input domain $\mathcal{D}^n$ is then partitioned into indistinguishable spaces $\{\simeq_v^f\}_{v \in \mathcal{D}}$.

When the adversary controls the parties $\mathsf{P_1}, \cdots, \mathsf{P_k}$, she will learn the set $\mathsf{Leak}_{\mathtt{iw}}^f(v, \overline{\mathbf{v}}_k) := \{(v_1, \cdots, v_n) \in \mathcal{D}^n \mid \overline{\mathbf{v}}_k = v_1, \cdots, v_k\} \cap \simeq_v^f$, from the result v and the adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$.

Definition 1 (Leakage in the ideal-world). *For an MPC application $f(\overline{\mathbf{x}}_n)$, the leakage of computing $v = f(\overline{\mathbf{v}}_n)$ in the ideal-world is $\mathsf{Leak}_{\mathtt{iw}}^f(v, \overline{\mathbf{v}}_k)$, for the adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$ and the result $v \in \mathcal{D}$.*

MPC in the real-world. An MPC application in the real-world is implemented using some MPC protocol π (denoted by π_f) by which all the parties collaboratively compute $\pi_f(\overline{\mathbf{x}})$ over their private inputs $\overline{\mathbf{v}}$ without any TTP T. Introduction of MPC protocols can be found in [14].

There are generally two types of adversaries in the real world, i.e., semi-honest and malicious. An adversary is semi-honest (a.k.a. passive) if the corrupted parties run the protocol honestly as specified, but may try to learn private information of other parties by observing the protocol execution (i.e., network messages and program states). An adversary is malicious (a.k.a. active) if the corrupted parties can deviate arbitrarily from the prescribed protocol (e.g., control, manipulate, and inject messages) in an attempt to learn private information of the other parties. In this work, we consider semi-honest adversaries, which are supported by most MPC frameworks and often serve as a basis for MPC in more robust settings with powerful adversaries.

A protocol π is (semi-honest) secure if what a (semi-honest) adversary can achieve in the real-world can also be achieved by a corresponding adversary in the ideal-world. Semi-honest security ensures that the corrupted parties learn no more information from executing the protocol than what they can learn from the result and the private inputs of the corrupted parties. Therefore, the leakage

of an MPC application $f(\overline{\mathbf{x}})$ in the real-world against the semi-honest adversary can also be characterized using the indistinguishability of private inputs.

Definition 2. *An MPC protocol π is (semi-honest) secure if for any MPC application $f(\overline{\mathbf{x}}_n)$, adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$ and result $v \in \mathcal{D}$, the leakage of computing $v = \pi_f(\overline{\mathbf{v}}_n)$ is $\mathsf{Leak}^f_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$.*

4 Language-level Leakage Characterization

In this section, we characterize the leakage of MPC applications from the language perspective.

4.1 A Language for MPC

We consider a simple language WHILE for implementing MPC applications. The syntax of WHILE programs is defined as follows.

$$\begin{aligned} p ::= & \mathsf{skip} \mid x = e \mid \ p_1; p_2 \mid \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \mid \mathsf{return}\ x \\ & \mid \mathsf{while}\ x\ \mathsf{do}\ p \mid \mathsf{repeat}\ n\ \mathsf{do}\ p \end{aligned}$$

where e is an expression defined as usual and n is a positive integer.

Despite its simplicity, WHILE suffices to illustrate our approach and our tool supports a real-world language. Note that we introduce two loop constructs. The while loop can only be used with the secret-independent conditions while the repeat loop (with a fixed number n of iterations) can have secret-dependent conditions. The restriction of the while loop is necessary, as the adversary knows when to terminate the loop, so secret information may be leaked if a secret-dependent condition is used [44].

The operational semantics of the WHILE program is defined in a standard way (cf. [15]). In particular, repeat n do p means repeating the loop body p for a fixed number n times. A configuration is a tuple $\langle p, \sigma \rangle$, where p denotes a statement and $\sigma : \mathcal{X} \to \mathcal{D}$ denotes a state that maps variables to values. The evaluation of an expression e under a state σ is denoted by $\sigma(e)$. A transition from $\langle p, \sigma \rangle$ to $\langle p', \sigma' \rangle$ is denoted by $\langle p, \sigma \rangle \to \langle p', \sigma' \rangle$ and $\to^*$ denotes the transitive closure of $\to$. An execution starting from the configuration $\langle p, \sigma \rangle$ is a sequence of configurations. We write $\langle p, \sigma \rangle \Downarrow \sigma'$ if $\langle p, \sigma \rangle \to^* \langle \mathsf{skip}, \sigma' \rangle$. We assume that each execution ends in a return statement, i.e., all the while loops always terminate. We denote by $\langle p, \sigma \rangle \Downarrow \sigma' : v$ the execution returning value v.

4.2 Leakage Characterization in Ideal/Real-World

An MPC application $f(\overline{\mathbf{x}})$ is implemented as a WHILE program p. An execution of the program p evaluates the computation $f(\overline{\mathbf{x}})$ as if a TTP directly executed the program p on the private inputs. In this setting, the adversary cannot observe any intermediate states of the execution other than the final result.

Let $\mathcal{X}^{\mathtt{in}} = \{x_1, \cdots, x_n\} \subseteq \mathcal{X}$ be the set of private input variables. We denote by State_0 the set of the initial states. Given a tuple of values $\overline{\mathbf{v}}_k \in \mathcal{D}^k$ and a result $v \in \mathcal{D}$, let $\mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ denote the set of states $\sigma \in \mathsf{State}_0$ such that $\langle p, \sigma\rangle \Downarrow \sigma' : v$ for some state σ' and $\sigma(x_i) = v_i$ for $1 \leq i \leq k$. Intuitively, when the adversary controls the parties $\mathsf{P}_1, \cdots, \mathsf{P}_k$, she learns the set of states $\mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ from the result v and the adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$. We can reformulate the leakage of an MPC application $f(\overline{\mathbf{x}})$ in the ideal-world (cf. Definition 1) as follows.

Proposition 1. *Given an MPC application $f(\overline{\mathbf{x}}_n)$ implemented by a program p, $\overline{\mathbf{v}}'_n \in \mathsf{Leak}^f_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ iff there exists a state $\sigma \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ such that $\sigma(x_i) = v'_i$ for $1 \leq i \leq n$.*

We use security policies to characterize the leakage of MPC applications in the real-world.

Security level. We consider a lattice of security levels $\mathbb{L} = \{\mathtt{Sec}, \mathtt{Pub}\}$ with $\mathtt{Pub} \sqsubseteq \mathtt{Pub}$, $\mathtt{Pub} \sqsubseteq \mathtt{Sec}$, $\mathtt{Sec} \sqsubseteq \mathtt{Sec}$ and $\mathtt{Sec} \not\sqsubseteq \mathtt{Pub}$. We denote by $\ell_1 \sqcup \ell_2$ the least upper bound of two security levels $\ell_1, \ell_2 \in \mathbb{L}$, namely, $\ell \sqcup \mathtt{Sec} = \mathtt{Sec} \sqcup \ell = \mathtt{Sec}$ for $\ell \in \mathbb{L}$ and $\mathtt{Pub} \sqcup \mathtt{Pub} = \mathtt{Pub}$.

Definition 3. *A security policy $\varrho : \mathcal{X} \to \mathbb{L}$ for the MPC application $f(\overline{\mathbf{x}})$ is a function that associates each variable $x \in \mathcal{X}$ with a security level $\ell \in \mathbb{L}$.*

Given a security policy ϱ and a security level $\ell \in \mathbb{L}$, let $\mathcal{X}^\ell := \{x \mid \varrho(x) = \ell\} \subseteq \mathcal{X}$, i.e., the set of variables with the security level ℓ under ϱ. We lift the order $\sqsubseteq$ to security policies, namely, $\varrho \sqsubseteq \varrho'$ if $\varrho(x) \sqsubseteq \varrho'(x)$ for each $x \in \mathcal{X}$. When executing the program p with a security policy ϱ using an MPC protocol π, we assume that the adversary can observe the values of the public variables $x \in \mathcal{X}^{\mathtt{Pub}}$, but not that of the secret variables $x \in \mathcal{X}^{\mathtt{Sec}}$.

This is a practical assumption and can be well-supported by the existing approach. For instance, Obliv-C [44] allows developers to define an MPC application in an extension of C language, when compiled and linked, the result will be a concrete garbled circuit protocol π_p whose computation does not reveal the values of any oblivious-qualified variables. Thus, all the secret variables specified by the security policy ϱ can be declared as oblivious-qualified variables in Obliv-C, while all the public variables specified by the security policy ϱ are declared without oblivious-qualification. Similarly, MPyC [37] is a Python package for implementing MPC applications that allows programmers to define instances of secret-typed variable classes using Python's class mechanism. When executing MPC applications, instances of secret-typed class variables are protected via Shamir's secret sharing protocol [38]. Thus, all the secret variables specified by the security policy ϱ can be declared as instances of secret-typed variable classes in MPyC, while all the public variables specified by the security policy ϱ are declared as instances of Python's standard classes.

Leakage under a security policy. Fix a security policy ϱ for the program p. Remark that the values of the secret variables will not be known even at runtime for each party, as they are encrypted. This means that, unlike the

secret-independent conditions, the secret-dependent conditions cannot be executed normally, and thus should be removed using, e.g., multiplexers, before transforming into circuits. We define the transformation $\mathcal{T}_\varrho(\cdot,\cdot)$, where c is the selector of a multiplexer.

$$\mathcal{T}_\varrho(c, p_1; p_2) \triangleq \mathcal{T}_\varrho(c, p_1); \mathcal{T}_\varrho(c, p_2) \qquad \mathcal{T}_\varrho(c, \mathsf{return}\ x) \triangleq \mathsf{return}\ x$$

$$\mathcal{T}_\varrho(c, x = e) \triangleq x = x + c \times (e - x) \qquad \mathcal{T}_\varrho(c, \mathsf{skip}) \triangleq \mathsf{skip}$$

$$\mathcal{T}_\varrho(c, \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2) \triangleq \begin{cases} \mathsf{if}\ x\ \mathsf{then}\ \mathcal{T}_\varrho(1, p_1)\ \mathsf{else}\ \mathcal{T}_\varrho(1, p_2), & \text{if } c = 1 \wedge \varrho(x) = \mathtt{Pub}; \\ \mathcal{T}_\varrho(c \& x, p_1); \mathcal{T}_\varrho(c \& \neg x, p_2), & \text{otherwise.} \end{cases}$$

$$\mathcal{T}_\varrho(c, \mathsf{while}\ x\ \mathsf{do}\ p) \triangleq \begin{cases} \mathsf{while}\ x\ \mathsf{do}\ \mathcal{T}_\varrho(1, p), & \text{if } c = 1 \wedge \varrho(x) = \mathtt{Pub}; \\ \mathtt{Error}, & \text{otherwise.} \end{cases}$$

$$\mathcal{T}_\varrho(c, \mathsf{repeat}\ n\ \mathsf{do}\ p) \triangleq \mathsf{repeat}\ n\ \mathsf{do}\ \mathcal{T}_\varrho(c, p)$$

Intuitively, c in $\mathcal{T}_\varrho(c,\cdot)$ indicates whether the statement is under some secret-dependent branching statements. Initially, $c = 1$. During the transformation, c will be conjuncted with the branching condition x or $\neg x$ when transforming $\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$ if x is secret or $c \neq 1$. The control flow inside should be protected if $c \neq 1$. If $c = 1$ and the condition variable x is public, the statement needs not be protected. $\mathcal{T}(c, x = e)$ simulates a multiplexer with two different values depending on whether the assignment $x = e$ is in the scope of some secret-dependent conditions. At runtime, the value e is assigned to x if c is 1, otherwise x does not change. $\mathcal{T}_\varrho(c, \mathsf{while}\ x\ \mathsf{do}\ p)$ enforces that the while loop is used in secret-independent conditions and x is public in the security policy ϱ otherwise throws an error. The other cases are trivial. We denote by $\widehat{p}_\varrho$ the program $\mathcal{T}_\varrho(1, p)$ on which we will define the leakage of p in the real-world.

For every state $\sigma : \mathcal{X} \to \mathcal{D}$, let $\sigma^{\mathtt{Pub}} : \mathcal{X}^{\mathtt{Pub}} \to \mathcal{D}$ denote the state that is the projection of the state σ onto the public variables $\mathcal{X}^{\mathtt{Pub}}$. For each execution $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow \sigma_2$, we denote by $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2$ the sequence of configurations where each state σ is replaced by the state $\sigma^{\mathtt{Pub}}$.

Recall that the adversary can observe the values of public variables $x \in \mathcal{X}^{\mathtt{Pub}}$ when executing the program $\widehat{p}_\varrho$. Thus, from an execution $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow \sigma_2 : v$, she can observe the sequence $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2$ and the result v, written as $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2 : v$. For every state $\sigma \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$, we denote by $\mathsf{Leak}^{p,\varrho}_{\mathtt{rw}}(v, \sigma)$ the set of states $\sigma' \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ such that $\langle \widehat{p}_\varrho, \sigma' \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma'_1 : v$ and $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_1 : v$ are identical.

Definition 4. *A security policy ϱ is perfect for a given MPC application $f(\overline{\mathbf{x}}_n)$ implemented by the program p, denoted by $\varrho \models_p f(\overline{\mathbf{x}}_n)$, if $\mathcal{T}_\varrho(1, p)$ does not throw any errors, and for adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$, the result $v \in \mathcal{D}$, and the state $\sigma \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$, we have that*

$$\mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k) = \mathsf{Leak}^{p,\varrho}_{\mathtt{rw}}(v, \sigma).$$

Intuitively, a perfect security policy ϱ ensures that for every state $\sigma \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$, from the observation $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma' : v$, the adversary only learns the same set $\mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ of initial states as that in the ideal-world.

Our goal is to compute a perfect security policy ϱ for every program p that implements the MPC $f(\overline{\mathbf{x}})$. A naive way is to assign the high security level $\mathtt{Sec}$ to all the variables $\mathcal{X}$, which may however suffer from a lower performance, as all the intermediate computations have to be performed on encrypted data and conditional statements have to removed. Ideally, a security policy ϱ should not only be perfect but also annotate as few secret variables as possible.

5 Type System

In this section, we present a sound type system to automatically infer perfect security policies. We first define noninterference of a program p w.r.t. a security policy ϱ, which is shown to entail the perfectness of ϱ.

Definition 5. *A program p is noninterfering w.r.t. a security policy ϱ, written as ϱ-noninterfering, if $\mathcal{T}_\varrho(1, p)$ does not throw any errors and $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2 : v$ and $\langle \widehat{p}_\varrho, \sigma_1' \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2' : v'$ are the same for each pair of states $\sigma_1, \sigma_1' \in \mathsf{State}_0$.*

Intuitively, the ϱ-noninterference ensures that for all private inputs of the n parties (without the adversary-chosen private inputs), the adversary observes the same sequence of the configurations from all the executions that return the same value.

The ϱ-noninterference of p entails the perfectness of ϱ where the adversary can choose arbitrary private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$ of the corrupted participants $(\mathsf{P}_1, \cdots, \mathsf{P}_k)$ for any $k \geq 1$.

Proposition 2. *If p is ϱ-noninterfering for a security policy ϱ, then $\varrho \models_p f(\overline{\mathbf{x}})$.*

Note that the converse of Proposition 2 does not necessarily hold due to the adversary-chosen private inputs. For instance, suppose $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2 : v$ and $\langle \widehat{p}_\varrho, \sigma_1' \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2' : v$ are identical for every pair of states $\sigma_1, \sigma_1' \in \mathsf{Leak}_{\mathtt{iw}}^p(v, v_1)$, and $\langle \widehat{p}_\varrho, \sigma_3 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_4 : v$ and $\langle \widehat{p}_\varrho, \sigma_3' \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_4' : v$ are identical for every pair of states $\sigma_3, \sigma_3' \in \mathsf{Leak}_{\mathtt{iw}}^p(v, v_1')$. If $v_1 \neq v_1'$, then $\langle \widehat{p}_\varrho, \sigma_1 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_2 : v$ and $\langle \widehat{p}_\varrho, \sigma_3 \rangle \Downarrow_\varrho^{\mathtt{Pub}} \sigma_4 : v$ are different, implying that p is not ϱ-noninterfering.

Based on Proposition 2, we present a type system for inferring a perfect security policy ϱ of a given program p such that p is ϱ-noninterfering. The typing judgement is in the form of $\mathsf{c} \vdash p : \varrho \Rightarrow \varrho'$, where the type contexts ϱ, ϱ' are security policies, p is the program under typing, and c is the security level of the current control flow. The typing judgement $\mathsf{c} \vdash p : \varrho \Rightarrow \varrho'$ states that given the security level of the current control flow c and the type context ϱ, the statement p is typable and yields a new updated type context ϱ'.

The type inference rules are shown in Fig. 3 which track the security levels of both data- and control-flow of information from private inputs, where $\varrho(e)$ denotes the least upper bound of the security levels $\varrho(x)$ of variables x used in the expression e and $\varrho_1 \sqcup \varrho_2$ is the security policy such that for every variable $x \in \mathcal{X}$, $(\varrho_1 \sqcup \varrho_2)(x) = \varrho_1(x) \sqcup \varrho_2(x)$. $\mathtt{lfp}(\mathsf{c}, n, \varrho, p)$ is ϱ if $n = 0$ or $\varrho' = \varrho$, otherwise $\mathtt{lfp}(\mathsf{c}, n-1, \varrho', p)$, where $\mathsf{c} \vdash p : \varrho \Rightarrow \varrho'$. Note that constants have the security level $\mathtt{Pub}$. Most of those rules are standard.

$$\frac{}{\mathsf{c} \vdash \mathsf{skip} : \varrho \Rightarrow \varrho}\,[\text{T-Skip}] \qquad \frac{\varrho' = \varrho[x \mapsto \mathsf{c} \sqcup \varrho(e)]}{\mathsf{c} \vdash x = e : \varrho \Rightarrow \varrho'}\,[\text{T-Assign}]$$

$$\frac{\mathsf{c} \vdash p_1 : \varrho \Rightarrow \varrho_1 \quad \mathsf{c} \vdash p_2 : \varrho_1 \Rightarrow \varrho_2}{\mathsf{c} \vdash p_1; p_2 : \varrho \Rightarrow \varrho_2}\,[\text{T-Seq}] \qquad \frac{\mathsf{c} \sqcup \varrho(x) \vdash p_1 : \varrho \Rightarrow \varrho_1 \quad \mathsf{c} \sqcup \varrho(x) \vdash p_2 : \varrho \Rightarrow \varrho_2 \quad \varrho' = \varrho_1 \sqcup \varrho_2}{\mathsf{c} \vdash \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : \varrho \Rightarrow \varrho'}\,[\text{T-If}]$$

$$\frac{}{\mathsf{c} \vdash \mathsf{return}\ x : \varrho \Rightarrow \varrho}\,[\text{T-Return}] \qquad \frac{\varrho' = \mathtt{lfp}(\mathsf{c}, n, \varrho, p)}{\mathsf{c} \vdash \mathsf{repeat}\ n\ \mathsf{do}\ p : \varrho \Rightarrow \varrho'}\,[\text{T-Repeat}]$$

$$\frac{\varrho(x) = \mathtt{Pub} \quad \mathsf{c} = \mathtt{Pub} \quad \varrho' = \mathtt{lfp}(\mathtt{Pub}, -1, \varrho, p)}{\mathsf{c} \vdash \mathsf{while}\ x\ \mathsf{do}\ p : \varrho \Rightarrow \varrho'}\,[\text{T-While}]$$

Fig. 3: Type inference rules

Rule T-Assign disables the data-flow and control-flow of information from the security level Sec to the security level Pub. To meet this constraint, the security level of the variable x is updated to the least upper bound $\mathsf{c} \sqcup \varrho(e)$ of the security levels of the current control flow c and variables used in the expression e. Rule T-If passes the security level c of the current control flow into both branches, preventing from assigning values to public variables in those two branches when $c = \mathtt{Sec}$. Rule T-While requires that the loop condition is public and the loop is used with secret-independent conditions, ensuring that $\mathcal{T}_\varrho(1, p)$ does not throw any errors. Rule T-Return does not impose any constraints on x, as the return value is observable to the adversary.

Let $\varrho_0 : \mathcal{X} \to \mathbb{L}$ be the mapping such that $\varrho_0(x) = \mathtt{Sec}$ for all $x \in \mathcal{X}^{\mathtt{Sec}}$, $\varrho_0(x) = \mathtt{Pub}$ otherwise. If the typing judgement $\mathtt{Pub} \vdash p : \varrho_0 \Rightarrow \varrho$ is valid, then the values of all the public variables specified by ϱ do not depend on any values of private inputs. Thus, it is straightforward to get that:

Proposition 3. *If the typing judgement* $\mathtt{Pub} \vdash p : \varrho_0 \Rightarrow \varrho$ *is valid, then the program* p *is* ϱ*-noninterfering.*

From Proposition 2 and Theorem 3, we have

Corollary 1. *If* $\mathtt{Pub} \vdash p : \varrho_0 \Rightarrow \varrho$ *is valid, then* ϱ *is perfect, i.e.,* $\varrho \models_p f(\overline{\mathbf{x}})$.

6 Degrading Security Levels

The type system allows to infer a security policy ϱ such that the type judgement $\mathtt{Pub} \vdash p : \varrho_0 \Rightarrow \varrho$ is valid, from which we can deduce that $\varrho \models_p f(\overline{\mathbf{x}})$, i.e., ϱ is perfect for the MPC application $f(\overline{\mathbf{x}})$ implemented by the program p. However, the security policy ϱ may be too conservative, i.e., some secret variables specified by ϱ can be declassified without compromising the security. In this section, we propose an automated approach to identify these variables. We mainly consider minimizing the number of secret branching variables, viz., the secret variables used in branching conditions, as they usually incur a high computation and communication overhead. W.l.o.g., we assume that for each secret branching variable x there is only one assignment to x and it is used only in one conditional

$$\frac{}{\lceil x = e, \alpha, \phi \rfloor \hookrightarrow \lceil \mathsf{skip}, \alpha[x \mapsto \alpha(e)], \phi \rfloor} \qquad \frac{}{\lceil \mathsf{return}\ x, \alpha, \phi \rfloor \hookrightarrow \lceil \mathsf{skip}, \alpha, \phi \rfloor}$$

$$\frac{\lceil p_1, \alpha_1, \phi_1 \rfloor \hookrightarrow \lceil \mathsf{skip}, \alpha_2, \phi_2 \rfloor \quad \lceil p_2, \alpha_2, \phi_2 \rfloor \hookrightarrow \lceil p_2', \alpha_3, \phi_3 \rfloor}{\lceil p_1; p_2, \alpha_1, \phi_1 \rfloor \hookrightarrow \lceil p_2', \alpha_3, \phi_3 \rfloor} \qquad \frac{\lceil p_1, \alpha_1, \phi_1 \rfloor \hookrightarrow \lceil p_1', \alpha_2, \phi_2 \rfloor \quad p_1' \neq \mathsf{skip}}{\lceil p_1; p_2, \alpha_1, \phi_1 \rfloor \hookrightarrow \lceil p_1'; p_2, \alpha_2, \phi_2 \rfloor}$$

$$\frac{\mathsf{SAT}(\phi') \quad \phi' = \phi \wedge \alpha(x)}{\lceil \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2, \alpha, \phi \rfloor \hookrightarrow \lceil p_1, \alpha, \phi' \rfloor} \qquad \frac{\mathsf{SAT}(\phi') \quad \phi' = \phi \wedge \neg\alpha(x)}{\lceil \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2, \alpha, \phi \rfloor \hookrightarrow \lceil p_2, \alpha, \phi' \rfloor}$$

$$\frac{\mathsf{SAT}(\phi') \quad \phi' = \phi \wedge \alpha(x) \quad p' = p; \mathsf{while}\ x\ \mathsf{do}\ p}{\lceil \mathsf{while}\ x\ \mathsf{do}\ p, \alpha, \phi \rfloor \hookrightarrow \lceil p', \alpha, \phi' \rfloor} \qquad \frac{\mathsf{SAT}(\phi') \quad \phi' = \phi \wedge \neg\alpha(x) \quad p' = \mathsf{skip}}{\lceil \mathsf{while}\ x\ \mathsf{do}\ p, \alpha, \phi \rfloor \hookrightarrow \lceil p', \alpha, \phi' \rfloor}$$

$$\frac{p' = (n \geq 1)\ ?\ p; \mathsf{repeat}\ n-1\ \mathsf{do}\ p\ :\ \mathsf{skip}}{\lceil \mathsf{repeat}\ n\ \mathsf{do}\ p, \alpha, \phi \rfloor \hookrightarrow \lceil p', \alpha, \phi \rfloor}$$

Fig. 4: The symbolic semantics of WHILE programs

statement. (We can rename variables in p if this assumption does not hold, where the named variables have the same security levels as their original names.) With this assumption, whether x can be declassified depends only on the unique conditional statement where it occurs.

Fix a security policy ϱ such that $\varrho \models_p f(\overline{\mathbf{x}})$. Suppose that $\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$ is not used with secret-dependent conditions. Let ϱ' be the security policy $\varrho[x \mapsto \mathtt{Pub}]$. It is easy to see that $\mathcal{T}_{\varrho'}(1, p)$ does not raise any errors. Therefore, to declassify x, we need to ensure that $\langle \widehat{p}_{\varrho'}, \sigma' \rangle \Downarrow^{\mathtt{Pub}}_{\varrho'} \sigma_1' : v$ and $\langle \widehat{p}_{\varrho'}, \sigma \rangle \Downarrow^{\mathtt{Pub}}_{\varrho'} \sigma_1 : v$ are identical for every adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$, result $v \in \mathcal{D}$, and states $\sigma, \sigma' \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$. However, as the number of the initial states may be large and even infinite, it is infeasible to check all pairs of executions.

We propose to use symbolic executions to represent the potentially infinite sets of (concrete) executions. Each symbolic execution t is associated with a path condition ϕ which denotes the set of initial states satisfying ϕ, from each of which the execution has the same sequence of statements. Thus, the conjunction $\phi \wedge e = v$, where e is the symbolic return value and v is concrete value, represents the set of initial states from which the executions have the same sequence of statements and returns the same result v. It is not difficult to observe that checking whether x in $\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$ can be declassified amounts to checking whether for every pair of symbolic executions t_1 and t_2 that both include $\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$, x has the same truth value in t_1 and t_2 whenever t_1 and t_2 return the same value. This can be solved by invoking off-the-shelf SMT solvers.

6.1 Symbolic Semantics

Let $\mathcal{E}$ denote the set of expressions over the private input variables $\overline{\mathbf{x}}$ and constants. A path condition $\phi \in \mathcal{E}$ is a conjunction of Boolean expressions. A state $\sigma \in \mathsf{State}_0$ satisfies ϕ, denoted by $\sigma \models \phi$, if ϕ evaluates to `True` under σ. A symbolic state α is a function $\mathcal{X} \to \mathcal{E}$ that maps variables to symbolic expressions. $\alpha(e)$ denotes the symbolic value of the expression e under α, obtained from e by replacing each occurrence of variable x by $\alpha(x)$. The initial symbolic state, denoted by α_0, is the identity function over the private input variables $\overline{\mathbf{x}}$.

The symbolic semantics of WHILE programs is defined by transitions between symbolic configurations, as shown in Fig. 4, where $\mathsf{SAT}(\phi)$ is $\texttt{True}$ iff the constraint ϕ is satisfiable. A symbolic configuration is a tuple $\lceil p, \alpha, \phi \rfloor$, where p is a statement, α is a symbolic state, and ϕ is the path condition that should be satisfied to reach $\lceil p, \alpha, \phi \rfloor$. $\lceil p, \alpha, \phi \rfloor \hookrightarrow \lceil p', \alpha', \phi' \rfloor$ denotes a transition from $\lceil p, \alpha, \phi \rfloor$ to $\lceil p', \alpha', \phi' \rfloor$. The symbolic semantics is almost the same as the operational semantics except that (1) the path conditions are collected and checked for conditional statements and while loops, and (2) the transition may be nondeterministic if both $\phi \wedge \alpha(x)$ and $\phi \wedge \neg\alpha(x)$ are satisfiable.

We denote by $\hookrightarrow^*$ the transitive closure of $\hookrightarrow$, where its path condition is the conjunction of that of each transition. An symbolic execution starting from a symbolic configuration $\lceil p, \alpha, \phi \rfloor$ is a sequence of symbolic configurations, written as $\lceil p, \alpha, \phi \rfloor \Downarrow (\alpha', \phi')$, if $\lceil p, \alpha, \phi \rfloor \hookrightarrow^* \lceil \mathsf{skip}, \alpha', \phi' \rfloor$. Moreover, we denote by $\lceil p, \alpha, \phi \rfloor \Downarrow (\alpha', \phi') : e$ the symbolic execution $\lceil p, \alpha, \phi \rfloor \Downarrow (\alpha', \phi')$ with the symbolic return value e. We denote by $\texttt{SymExe}$ the set of all the symbolic executions $\lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha, \phi) : e$ of the program p. Note that α_0 is the initial symbolic state. Recall that we assumed all the (concrete) executions always terminate, thus $\texttt{SymExe}$ is a finite set of finite sequence of symbolic configurations.

6.2 Relating Symbolic Executions to Concrete Executions

A symbolic execution $t = \lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha, \phi) : e$ represents the set of (concrete) executions starting from the states $\sigma \in \mathsf{State}_0$ such that $\sigma \models \phi$. Formally, consider $\sigma \in \mathsf{State}_0$ such that $\sigma \models \phi$, by concretizing all the symbolic values of variables x in each symbolic state α' with concrete values $\sigma(\alpha'(x))$ and projecting out all the path conditions, the symbolic execution t is the execution $\langle p, \sigma \rangle \Downarrow \sigma' : \sigma(e)$, written as $\sigma(t)$. For the execution $\langle p, \sigma \rangle \Downarrow \sigma' : v$, there are a unique symbolic execution t such that $\sigma(t) = \langle p, \sigma \rangle \Downarrow \sigma' : v$ and a unique execution $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow \sigma' : v$ in the program $\widehat{p}_\varrho$. We denote by $\mathsf{RW}_{\varrho,\sigma}(t)$ the execution $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow_\varrho \sigma' : v$ and denote by $\mathsf{RW}^{\mathtt{Pub}}_{\varrho,\sigma}(t)$ the sequence $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow^{\mathtt{Pub}}_\varrho \sigma' : v$.

For every adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$, result $v \in \mathcal{D}$, and initial state $\sigma \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$, we can reformulate the set $\mathsf{Leak}^{p,\varrho}_{\mathtt{rw}}(v, \sigma)$ as follows. (Recall that $\mathsf{Leak}^{p,\varrho}_{\mathtt{rw}}(v, \sigma)$ is the set of states $\sigma' \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ such that $\langle \widehat{p}_\varrho, \sigma' \rangle \Downarrow^{\mathtt{Pub}}_\varrho \sigma'_1 : v$ and $\langle \widehat{p}_\varrho, \sigma \rangle \Downarrow^{\mathtt{Pub}}_\varrho \sigma_1 : v$ are identical.)

Proposition 4. *For each state $\sigma' \in \mathsf{Leak}^p_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$, $\sigma' \in \mathsf{Leak}^{p,\varrho}_{\mathtt{rw}}(v, \sigma)$ iff for every symbolic execution $t' = \lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha', \phi') : e' \in \texttt{SymExe}$ such that $\sigma' \models \phi' \wedge e' = v$, $\mathsf{RW}^{\mathtt{Pub}}_{\varrho,\sigma}(t)$ and $\mathsf{RW}^{\mathtt{Pub}}_{\varrho,\sigma'}(t')$ are identical, where t is a symbolic execution $\lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha, \phi) : e$ such that $\sigma \models \phi \wedge e = v$.*

Proposition 4 allows to consider only the symbolic executions $\lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha, \phi) : e \in \texttt{SymExe}$ such that $\sigma \models \phi \wedge e = v$ when checking if ϱ is perfect or not.

6.3 Reasoning about Symbolic Executions

We leverage Proposition 4 to identify secret variables that can be declassified without compromising the security by reasoning about symbolic executions. For

each expression $\phi \in \mathcal{E}$, $\mathsf{Primed}(\phi)$ denotes the "primed" expression ϕ where each private input variable x_i is replaced by x_i' (i.e., its primed version).

Consider two symbolic executions $t = \lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha, \phi) : e$ and $t' = \lceil p, \alpha_0, \texttt{True} \rfloor \Downarrow (\alpha', \phi') : e'$. Assume if x then p' else p'' is not used with any secret-dependent conditions. Recall that we assumed x is used only in if x then p' else p''. Then, t and t' execute the same subsequence (say $p_1, \cdots, p_m$) of the statements that are if x then p' else p''. Let $e_1, \cdots, e_m$ (resp. $e_1', \cdots, e_m'$) be symbolic values of x when executing $p_1, \cdots, p_m$ in the symbolic execution t (resp. t'). Define the constraint $\Psi_x(t, t')$ as

$$\Psi_x(t,t') \triangleq \big(\phi \wedge \mathsf{Primed}(\phi') \wedge e = \mathsf{Primed}(e')\big) \Rightarrow \big(\textstyle\bigwedge_{i=1}^{m} e_i = \mathsf{Primed}(e_i')\big)$$

Intuitively, $\Psi_x(t, t')$ asserts that for every pair of states $\sigma, \sigma' \in \mathsf{State}_0$ if σ (resp. σ') satisfies the path condition ϕ (resp. ϕ'), $\sigma(e)$ and $\sigma'(e')$ are identical, then for each $1 \leq i \leq m$, the values of x are the same when executing the conditional statement p_i in both $\mathsf{RW}_{\varrho,\sigma}(t)$ and $\mathsf{RW}_{\varrho,\sigma'}(t')$.

Proposition 5. *For each pair of states $\sigma, \sigma' \in \mathsf{Leak}^{p}_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$ such that $\sigma \models \phi \wedge e = v$ and $\sigma' \models \phi' \wedge e' = v$, if $\Psi_x(t, t')$ is valid and $\mathsf{RW}^{\mathtt{Pub}}_{\varrho,\sigma}(t)$ and $\mathsf{RW}^{\mathtt{Pub}}_{\varrho,\sigma'}(t')$ are identical, then $\mathsf{RW}^{\mathtt{Pub}}_{\varrho',\sigma}(t)$ and $\mathsf{RW}^{\mathtt{Pub}}_{\varrho',\sigma'}(t')$ are identical, where $\varrho' = \varrho[x \mapsto \mathtt{Pub}]$.*

Recall that x can be declassified in a perfect security policy ϱ if $\varrho' = \varrho[x \mapsto \mathtt{Pub}]$ is still perfect, namely, $\langle \widehat{p}_{\varrho'}, \sigma' \rangle \Downarrow^{\mathtt{Pub}}_{\varrho'} \sigma_1' : v$ and $\langle \widehat{p}_{\varrho'}, \sigma \rangle \Downarrow^{\mathtt{Pub}}_{\varrho'} \sigma_1 : v$ are identical for every adversary-chosen private inputs $\overline{\mathbf{v}}_k \in \mathcal{D}^k$, result $v \in \mathcal{D}$, and states $\sigma, \sigma' \in \mathsf{Leak}^{p}_{\mathtt{iw}}(v, \overline{\mathbf{v}}_k)$. By Proposition 5, if $\Psi_x(t, t')$ is valid for each pair of symbolic executions $t, t' \in \mathtt{SymExe}$, we can deduce that ϱ' is still perfect.

Theorem 1. *If $\varrho \models_p f(\overline{\mathbf{x}})$ and $\Psi_x(t, t')$ is valid for each pair of symbolic executions $t, t' \in \mathtt{SymExe}$, then $\varrho[x \mapsto \mathtt{Pub}] \models_p f(\overline{\mathbf{x}})$.*

Example 1. Consider two symbolic executions t and t' in the motivating example such that the path condition ϕ (resp. ϕ') of t (resp. t') is $\mathtt{a} \geq \mathtt{b} \wedge \mathtt{c} > \mathtt{a}$ (resp. $\mathtt{a} < \mathtt{b} \wedge \mathtt{c} > \mathtt{b}$), and both return the result 3. The secret branching variable `c2` has the symbolic values $\mathtt{c} > \mathtt{a}$ (resp. $\mathtt{c} > \mathtt{b}$) in t and t', respectively. Then

$$\Psi_{\mathtt{c2}}(t,t') \triangleq (\mathtt{a} \geq \mathtt{b} \wedge \mathtt{c} > \mathtt{a} \wedge \mathtt{a}' < \mathtt{b}' \wedge \mathtt{c}' > \mathtt{b}' \wedge 3 = 3) \Rightarrow ((\mathtt{c} > \mathtt{a}) = (\mathtt{c}' > \mathtt{b}')).$$

Obviously, $\Psi_{\mathtt{c2}}(t, t')$ is valid. We can show that for any other pair (t, t') of symbolic executions, $\Psi_{\mathtt{c2}}(t, t')$ is always valid. Therefore, the secret branching variable `c2` can be declassified in any perfect security policy ϱ.

In contrast, the secret branching variable `c1` has the symbolic value $\mathtt{a} < \mathtt{b}$ in both t and t'. Then,

$$\Psi_{\mathtt{c1}}(t,t') \triangleq (\mathtt{a} \geq \mathtt{b} \wedge \mathtt{c} > \mathtt{a} \wedge \mathtt{a}' < \mathtt{b}' \wedge \mathtt{c}' > \mathtt{b}' \wedge 3 = 3) \Rightarrow ((\mathtt{a} < \mathtt{b}) = (\mathtt{a}' < \mathtt{b}')).$$

$\Psi_{\mathtt{c1}}(t, t')$ is not valid, thus the secret branching variable `c1` cannot be declassified.

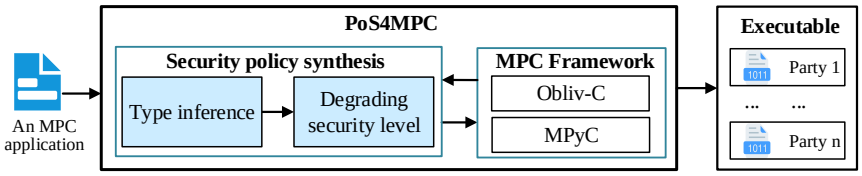

Fig. 5: The workflow of our tool **PoS4MPC**

Refinement. Theorem 1 allows us to check if the secret branching variable x of a conditional statement $\mathsf{if}\ x\ \mathsf{then}\ p'\ \mathsf{else}\ p''$ that does not used with any secret-dependent conditions can be declassified. After that, if x can be declassified without compromising the security, we feed back the result to the type system before checking the next secret branching variable. This allows us to refine the security level of variables that are updated in branches, namely, the type inference rule T-If is refined to the following one

$$\frac{\mathsf{c}' = (\text{can } x \text{ be declassified } ?\ \mathtt{Pub} : \varrho(x)) \qquad \mathsf{c} \sqcup \mathsf{c}' \vdash p_1 : \varrho \Rightarrow \varrho_1 \quad \mathsf{c} \sqcup \mathsf{c}' \vdash p_2 : \varrho \Rightarrow \varrho_2 \quad \varrho' = \varrho_1 \sqcup \varrho_2}{\mathsf{c} \vdash \mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : \varrho \Rightarrow \varrho'} \quad [\text{T-If}]$$

7 Implementation and Evaluation

We have implemented our approach in a tool, named **PoS4MPC**. The workflow of **PoS4MPC** is shown in Fig. 5, The input is an MPC program in C, which is parsed to an intermediate representation (IR) inside the LLVM Compiler [1] where call graph and control flow graphs are constructed at the LLVM IR level. We then perform the type inference which computes the a perfect security policy for the given program. To be accurate, we perform a field-sensitive pointer analysis [6] and our type inference is also field-sensitive. As the next step, we leverage the KLEE symbolic execution engine [10] to explore all the feasible symbolic executions, as well as the symbolic values of the return variable and secret branching variables of each symbolic execution. We fully explore loops since the bounds of loops in MPC are public and decided by user-specified inputs. Based on them, we iteratively check if a secret branching variable is degraded and the result is fed back to the type inference to refine security levels before checking the next secret branching variable. After that, we transform the program into the input of Obliv-C [44] by which the program can be compiled into executable implementations, one for each party. Obliv-C is an extension of C for implementing 2-party MPC applications using Yao's garbled circuit protocol [43]. For experimental purposes, **PoS4MPC** also features the high-level MPC framework MPyC [37], which is a Python package for implementing n-party MPC applications ($n \geq 1$) using Shamir's secret sharing protocol [38]. The C program is transformed into Python by a translator.

We also implement an optimization in our tool to alleviate the path explosion problem. Instead of directly checking the validity of $\Psi_x(t, t')$ for each secret branching variable x and pair of symbolic executions t and t', we first check if

Table 2: Number of (secret) branching variables

Name	LOC	#Branch var	#Other var	#Secret branch var		#Other secret var	
				After TS	After Check	Before refinement	After refinement
QS	56	4	6	3	0	4	2
LinS	25	1	3	1	0	2	1
BinS	46	2	8	2	1	6	6
AlmS	73	6	10	6	4	8	8
PSI	34	1	5	1	0	3	1

the premise $\phi \wedge \mathsf{Primed}(\phi') \wedge e = \mathsf{Primed}(e')$ of $\Psi_x(t, t')$ is satisfiable. We can conclude that $\Psi_x(t, t')$ is valid for any secret branching variable x if the premise $\phi \wedge \mathsf{Primed}(\phi') \wedge e = \mathsf{Primed}(e')$ is unsatisfiable. Furthermore, this yields a sound compositional reasoning approach which allows to split a program into a sequence of function calls. When each pair of the symbolic executions for each function cannot result in the same return value, we can conclude that $\Psi_x(t, t')$ is valid for any secret branching variable x and any pair of symbolic executions t and t' of the entire program. This optimization reduces the evaluation time of symbolic execution of `PSI` (resp. `QS`) from 95.9s–8.1h (resp. 504.6s) to 1.7s–79.6s (resp. 11.6s) in input array size varies from 10 to 100 (resp. 10).

7.1 Evaluation Setup

For an evaluation of our approach, we conduct experiments on five typical 2-party MPC applications [2], i.e., quicksort (`QS`) [21], linear search (`LinS`) [13], binary search (`BinS`) [13], almost search (`AlmS`), and private set intersection (`PSI`) [5]. `QS` outputs the list of indices of a given integer array $\overline{\mathbf{a}}$ in its ordered version, where the first half of $\overline{\mathbf{a}}$ is given by one party and the second half of $\overline{\mathbf{a}}$ is given by the another party. `LinS` (resp. `BinS` and `AlmS`) outputs the index of an integer b in an array $\overline{\mathbf{a}}$ if it exists, -1 otherwise, where the integer array $\overline{\mathbf{a}}$ is the input from one party and the integer b is the input from the another party. `LinS` always scans the array from the start to the end even though it has found the integer b. `BinS` is a standard iterative approach on a sorted array, where the array index is protected via oblivious read access machine [20]. `AlmS` is a variant of `BinS`, where the input array is almost sorted, namely, each element is at either the correct position or the closest neighbour of the correct position. `PSI` outputs the intersection of two integer sets, each of which is an input from one party.

All the experiments were conducted on a desktop with 64-bit Linux Mint 20.1, Intel Core i5-6300HQ CPU, 2.30 GHz and 8 GB RAM. When evaluating MPC applications, the client of each party is executed with a single thread.

7.2 Performance of Security Policy Synthesis

Security policy. The results of our approach is shown in Table 2, where column (LOC) shows the number of lines of code, column (#Branch var) shows the number of branching variables while column (#Other var) shows the number of other variables, columns (After TS) and (After Check) respectively show the

Table 3: Execution time of our security policy synthesis approach

Name	Length																			
	10		20		30		40		50		60		70		80		90		100	
	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check	SE	Check
QS	11.6	0.8	0.4h	304.2	2.0h	959.8	5.0h	0.6h	9.5h	0.9h	15.5h	1.3h	22.6h	1.6h	31.0h	2.0h	40.7h	2.3h	51.6h	2.7h
LinS	0.4	1.0	0.6	1.0	1.0	1.0	1.4	1.0	2.0	1.1	2.6	1.1	3.4	1.2	4.2	1.2	5.2	1.3	6.2	1.4
BinS	0.8	1.1	2.1	4.3	3.8	10.2	6.4	20.0	9.5	34.8	13.8	54.6	19.5	80.1	25.6	103.4	34.1	151.4	42.7	204.7
AlmS	1.3	0.8	4.3	3.5	7.7	10.0	14.1	18.6	20.6	32.3	28.9	51.0	40.7	77.4	55.1	110.3	74.9	148.2	94.4	200.0
PSI	1.7	0.5	4.3	1.0	8.0	1.5	13.2	2.1	20.0	2.8	28.6	3.5	39.3	4.3	50.9	5.3	63.0	6.4	79.6	7.8

number of secret branching variables after applying the type system and checking if the secret branching variables can be declassified, columns (Before refinement) and (After refinement) respectively show the number of other secret variables before and after refining the type inference by feeding back the results of the symbolic reasoning. (Note that the input variables are excluded in counting.)

We can observe that only few variables (2 for `QS`, 1 for `LinS`, 2 for `BinS`, 2 for `AlmS` and 2 for `PSI`) can be found to be public by solely using the type system. With our symbolic reasoning approach, more secret branching variables can be declassified without compromising the security (3 for `QS`, 1 for `LinS`, 1 for `BinS`, 2 for `AlmS` and 1 for `PSI`). After refining the type inference using results of the symbolic reasoning approach, more secret variables can be declassified (2 for `QS`, 1 for `LinS` and 2 for `PSI`). Overall, our approach annotates 2, 1, 7, 12 and 1 internal variables as secret out of 10, 4, 10, 16 and 6 variables for `QS`, `LinS`, `BinS`, `AlmS` and `PSI`, respectively.

Execution time. The execution time of our approach is shown in Table 3, where columns (SE) and (Check) respectively show the execution time (in second unless indicated by h for hour) of collecting symbolic executions and checking if secret branching variables can be declassified, by varying the size of the input array for each program from 10 to 100 with step 10. We did not report the execution time of our type system, as it is less than 0.1 second for each benchmark.

We can observe that our symbolic reasoning approach is able to check all the secret branching variables in few minutes (up to 294.4s) except for `QS`. After an in-depth analysis, we found that the number of symbolic executions is exponential in the length of the input array for `QS` and `PSI` while it is linear in the length of the input array for the other benchmarks. Our compositional reasoning approach works very well on `PSI`, otherwise it would take similar execution time as on `QS`. Indeed, a loop of `PSI` is implemented as a sequence of function calls each of which has a fixed number of symbolic executions. Furthermore, each pair of symbolic executions in the called function cannot result in the same return value. Therefore, the number of symbolic executions and the execution time of our symbolic reasoning approach is reduced significantly. However, our compositional reasoning approach does not work on `QS`. Although the number of symbolic executions grows exponentially on `QS`, the execution time of checking if secret branching variables can be declassified is still reduced by our optimization, which avoids the checking of the constraint $\Psi_x(t, t')$ if its premise $\phi \wedge \mathsf{Primed}(\phi') \wedge e = \mathsf{Primed}(e')$ is unsatisfiable.

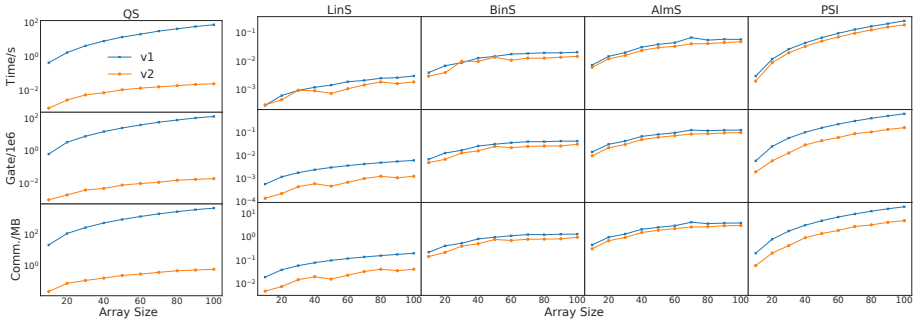

Fig. 6: Execution time (Time) in second, the number of gates (Gate) in 10^6 gates, Communication (Comm.) in MB using Obliv-C

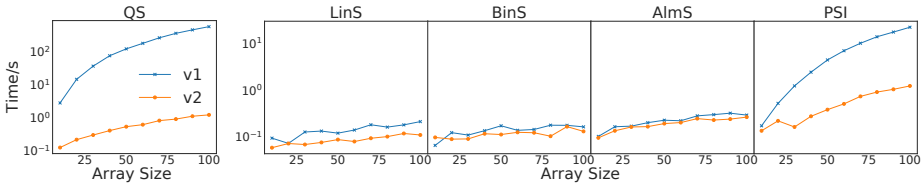

Fig. 7: Execution time (Time) in second using MPyC

7.3 Performance Improvement of MPC Applications

To evaluate the performance improvement of the MPC applications, we compare the execution time (in second), the size of the circuits (in $10^6\times$gates), and the volume of communication traffic (in MB) of each benchmark with the security policies v1 and v2, where v1 is obtained by solely applying our type system and v2 is obtained from v1 by degrading security levels and refinement without compromising the security. The measurement results are calculated by $\frac{\text{result of v1}}{\text{result of v2}} - 1$, taking the average of 10 times repetitions in order to minimize the noise.

Obliv-C. The results in Obliv-C are depicted in Fig. 6 (note the logarithmic scale of the vertical coordinate), where the size of the random input array for each benchmark varies from 10 to 100 with step size 10. Overall, we can observe that the performance improvement is significant especially on `QS`. In detail, compared with the security policy v1 on `QS` (resp. `LinS`, `BinS`, `AlmS`, and `PSI`), on average the security policy v2 reduces (1) the execution time by $1.56 \times 10^5\%$ (resp. 45%, 38%, 31% and 36%), (2) the size of circuits by $3.61\times10^5\%$ (resp. 368%, 52%, 38% and 275%), and (3) the volume of communication traffic by $4.17 \times 10^5\%$ (resp. 367%, 53%, 39% and 274%). This demonstrates the performance improvement of the MPC applications in Obliv-C that uses Yao's garbled circuit protocol.

MPyC. The results in MPyC are depicted in Fig. 7. Since MPyC does not provide the size of circuits and the volume of communication traffic, we only report execution time in Fig. 7. The results show that degrading security levels also improves execution time in MPyC that uses Shamir's secret sharing protocol. Compared with the security policy v1 on benchmark `QS` (resp. `LinS`, `BinS`, `AlmS`,

and PSI), on average the security policy v2 reduces the execution time by 2.5×10^4% (resp. 64%, 23%, 17% and 996%).

We note the difference in improvements of Obliv-C and MPyC. It is because: (1) Obliv-C and MPyC use different MPC protocols with varying improvements, where Yao's protocol (Obliv-C) is efficient for Boolean computations while the secret-sharing protocol (MPyC) is efficient for arithmetic computations; and (2) the proportion of downgrading variables is different where a larger proportion of downgrading variables (in particular branching variables with large branches) boosts performance more.

8 Related work

MPC Frameworks. Early efforts to MPC frameworks provide high-level languages for specifying MPC applications and compilers for translating them into executable implementations [31,8,23,32]. For instance, Fairplay complies 2-party MPC programs written in a domain-specific language into Yao's garbled circuits [31]. FairplayMP [8] extends Fairplay to multi-party using a modified version of the BMR protocol [7] with a Java interface. The others are aimed at improving the efficiency of operations in circuits and size of circuits. Mixed MPC protocols were also proposed to improve efficiency [26,9,34], as the efficiency of MPC protocols vary in operations. These frameworks explore the implementation space of operations in specific MPC protocols (e.g., garbled circuits, secret sharing and homomorphic encryption), as well as their conversions. However, all these frameworks either entirely compile an MPC program or compile an MPC program according to user-annotated secret variables to improve performance without formal security guarantees. Our approach improves the performance of MPC applications by declassifying secret variables without compromising security, which is orthogonal to the above optimization work.

Security of MPC applications. Since MPC applications implemented in MPC frameworks are not necessarily secure due to information leakage during execution in the real-world. Therefore, information-flow type systems and data-flow analysis have been adopted in the MPC frameworks, e.g., [44,37,24]. However, they only consider security verification but not automatic generation of security policies as we did in the current paper. Moreover, these approaches cannot identify some variables (e.g., c2 in our motivating example) that can actually be declassified without compromising security. Kerschbaum [25] proposed to infer public intermediate values by reasoning about epistemic modal logic, with a similar goal to ours for declassifying secret variables. However, it is unclear how efficient this approach is, as the performance of their approach was not reported [25].

Alternatively, self-composition which reduces the security problem to the safety problem on two copies of a program has been adopted by [3], where the safety problem can be solved by safety verification tools. However, safety verification remains challenging and these approaches often require user annotations (e.g., procedure contracts and loop invariants) that are non-trivial for MPC

practitioners. Our work is different from them in: (1) they only use the self-composition reduction to verify security instead of automatically generating a security policy; (2) they have to check almost all the program variables which is computational expensive, while we first apply an efficient type system to infer a security policy and then only check if the security branching variables in the security policy can be declassified; and (3) we check if security branching variables can be declassified by reasoning about pairs of symbolic executions which can be seen as a divide-and-conquer approach without annotations, and the results can be fed back to the type system to efficiently refine security levels. We remark that the self-composition reduction could also be used to check if a security branching variable could be declassified.

Information-flow analysis. A rich body of literature has studied verification of information-flow security and noninterference in programs [12], which requires that confidential data does not flow to outputs. This is too restrictive for programs which allow secret data to flow to some non-secret outputs, e.g., MPC applications, therefore the security notion is extended with declassification (a.k.a. delimited release) later [27]. These security problems are verified by type systems (e.g. [27]) or self-composition (e.g., [39]) or relational reasoning (e.g., [4]). Some of these techniques have been adapted to verify timing side-channel security, e.g., [11,42,30]. However, as the usual notions of security in these settings do not require reasoning about arbitrary leakage, these techniques are not directly applicable to our setting. Different from existing analysis using symbolic execution[33], our approach takes advantage of the public outputs of MPC programs and regards the public outputs as a part of leakage to avoid false positive of the noninterference approach and the quantification of information flow.

Finally, we remark that the leakage model considered in this work is different from the ones used in power side-channel security [45,18,19,17,16] and timing side-channel security [36,11,42,30] which leverage side-channel information while ours assumes that the adversary is able to observe all the public information during computation.

9 Conclusion

We have formalized the leakage of an MPC application which bridge the language-level and protocol-level leakages via security policies. Based on the formalization, we have presented an approach to automatically synthesize a security policy which can improve the performance of MPC applications while not compromising their privacy. Our approach is essentially a synergistic integration of type inference and symbolic reasoning with security type refinement. We implemented our approach in a tool **PoS4MPC**. The experimental results on five typical MPC applications confirm that our approach can significantly improve the performance of MPC applications.

References

1. The LLVM compiler infrastructure. https://llvm.org
2. The source code of our benchmarks. https://github.com/SPoS4/PoS4MPC (2022)
3. Almeida, J.B., Barbosa, M., Barthe, G., Pacheco, H., Pereira, V., Portela, B.: Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks. In: CSF. pp. 132–146 (2018)
4. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL. pp. 91–102 (2006)
5. Andreea, I.: Private set intersection: Past, present and future. In: SECRYPT. pp. 680–685 (2021)
6. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for C and C++. In: SAS. pp. 84–104 (2016)
7. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: STOC. pp. 503–513 (1990)
8. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS. pp. 257–266 (2008)
9. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: HyCC: Compilation of hybrid protocols for practical secure computation. In: CCS. pp. 847–861 (2018)
10. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224 (2008)
11. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: CCS. pp. 875–890 (2017)
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
13. Doerner, J.: The absentminded crypto kit. https://bitbucket.org/jackdoerner/absentminded-crypto-kit/
14. Evans, D., Kolesnikov, V., Rosulek, M.: A pragmatic introduction to secure multi-party computation. Found. Trends Priv. Secur. **2**(2-3), 70–246 (2018)
15. Fan, Y., Song, F., Chen, T., Zhang, L., Liu, W.: Pos4mpc: Automated security policy synthesis for secure multi-party computation. Technical report, https://faculty.sist.shanghaitech.edu.cn/faculty/songfu/publications/CAV22full.pdf, ShanghaiTech University (2022)
16. 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), 26:1–26:42 (2021)
17. 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)
18. Gao, P., Xie, H., Zhang, J., Song, F., Chen, T.: Quantitative verification of masked arithmetic programs against side-channel attacks. In: TACAS. pp. 155–173 (2019)
19. 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), 16:1–16:32 (2019)
20. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
21. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: ICISC. vol. 7839, pp. 202–216 (2012)

22. Hemenway, B., Lu, S., Ostrovsky, R., IV, W.W.: High-precision secure computation of satellite collision probabilities. In: SCN. pp. 169–187 (2016)
23. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: CCS. pp. 772–783 (2012)
24. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: CCS. pp. 1575–1590 (2020)
25. Kerschbaum, F.: Automatically optimizing secure computation. In: CCS. pp. 703–714 (2011)
26. Laud, P., Randmets, J.: A domain-specific language for low-level secure multiparty computation protocols. In: CCS. pp. 1492–1503 (2015)
27. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: POPL. pp. 158–170 (2005)
28. Lindell, Y.: Secure multiparty computation. Commun. ACM **64**(1), 86–96 (2021)
29. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: A programming framework for secure computation. In: S&P. pp. 359–376 (2015)
30. Malacaria, P., Khouzani, M.H.R., Pasareanu, C.S., Phan, Q., Luckow, K.S.: Symbolic side-channel analysis for probabilistic programs. In: CSF. pp. 313–327 (2018)
31. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security Symposium. pp. 287–302 (2004)
32. Mood, B., Gupta, D., Carter, H., Butler, K.R.B., Traynor, P.: Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In: EuroS&P. pp. 112–127 (2016)
33. Pasareanu, C.S., Kersten, R., Luckow, K.S., Phan, Q.: Chapter six - symbolic execution and recent applications to worst-case execution, load testing, and security analysis. Adv. Comput. **113**, 289–314 (2019)
34. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: improved mixed-protocol secure two-party computation. In: USENIX Security Symposium. pp. 2165–2182 (2021)
35. Poddar, R., Kalra, S., Yanai, A., Deng, R., Popa, R.A., Hellerstein, J.M.: Senate: A maliciously-secure MPC platform for collaborative analytics. In: USENIX Security Symposium. pp. 2129–2146 (2021)
36. Qin, Q., JiYang, J., Song, F., Chen, T., Xing, X.: Preventing timing side-channels via security-aware just-in-time compilation. CoRR **abs/2202.13134** (2022)
37. Schoenmakers, B.: MPyC: Secure multiparty computation in python. `https://github.com/lschoe/mpyc` (2020)
38. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
39. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: SAS. pp. 352–367 (2005)
40. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
41. Wagh, S., Gupta, D., Chandran, N.: Securenn: Efficient and private neural network training. IACR Cryptol. ePrint Arch. p. 442 (2018)
42. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: CAV. pp. 136–156 (2018)
43. Yao, A.C.: Protocols for secure computations. In: FOCS. pp. 160–164 (1982)
44. Zahur, S., Evans, D.: Obliv-C: A language for extensible data-oblivious computation. IACR Cryptol. ePrint Arch. p. 1153 (2015)
45. Zhang, J., Gao, P., Song, F., Wang, C.: SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In: CAV. pp. 157–177 (2018)