Knowledge Inference for Optimizing Secure Multi-party Computation

Aseem Rastogi Piotr Mardziel Michael Hicks Matthew Hammer

University of Maryland, College Park

Abstract

In secure multi-party computation, mutually distrusting parties cooperatively compute functions of their private data; in the process, they only learn certain results as per the protocol (e.g., the final output). The realization of these protocols uses cryptographic techniques to avoid leaking information between the parties. A protocol for a secure computation can sometimes be optimized without changing its security guarantee: when the parties can use their private data and the revealed output to infer the values of other data, then this other data need not be concealed from them via cryptography.

In the context of automatically optimizing secure multi-party computation, we define two related problems, *knowledge inference* and *constructive knowledge inference*. In both problems, we attempt to automatically discover when and if intermediate variables used in a protocol will (eventually) be known to the parties involved in the computation. Provably-known variables offer optimization opportunities.

We formally state the problem of knowledge inference (and its constructive variant); we describe our solutions, which are built atop existing, standard technology such as SMT solvers. We show that our approach is sound, and further, we characterize the completeness properties enjoyed by our approach. We have implemented our approach, and present a preliminary experimental evaluation.

1. Introduction

Secure multi-party computation (SMC) protocols [5, 10, 22] enable a number of parties $p_1, ..., p_n$ to cooperatively compute a function f over their private inputs $x_1, ..., x_n$ in a way that every party directly sees only the output $f(x_1, ..., x_n)$ while keeping the variables x_i private. Initial implementations of SMC protocols [6, 17] compute f using a single, monolithic protocol, which can be very expensive. More recently, researchers have been exploring how program analysis can be use to optimize a SMC.

A key observation underlying many optimizations is that while SMC protocols typically only reveal the final output to each party, a party may be able to *infer* the results of intermediate computations given the final output, their inputs, and the function being computed. When such inference is possible, the inferable intermediate results need not be cryptographically concealed. Revealing inferable results does not change the *knowledge profile* of the protocol: If the party will eventually know the intermediate result (e.g., given the final output), then revealing it earlier does not change what is known to whom.[1] Beyond preserving a protocol's knowledge profile, decomposing monolithic protocols into smaller protocols that explicitly share intermediate results can significantly improve their performance.

```
## assume a1 < a2, b1 < b2, distinct(a1, a2, b1, b2)
int median(int a1, int a2, int b1, int b2)
  bool x1, x2; int a3, b3, m;

  x1 = a1 ≤ b1;
  if x1 then { a3 = a2; } else { a3 = a1; }
  if x1 then { b3 = b1; } else { b3 = b2; }
  x2 = a3 ≤ b3;
  if x2 then { m = a3; } else { m = b3; }
  return m;
```

Figure 1. Joint median computation example [14]. `a1` and `a2` are Alice's inputs and `b1` and `b2` are Bob's. Both Alice and Bob can infer `x1` and `x2` given the final output.

As an example, consider the joint median computation between two parties Alice and Bob in Figure 1. Let `a1` and `a2` be Alice's inputs and `b1` and `b2` be Bob's. We also assume that these numbers are distinct, with `a1` $<$ `a2` and `b1` $<$ `b2`. At the end of the computation, both parties share the joint median output `m`.

In the unoptimized version of secure computation for this example, the whole program is computed as a single secure computation. However, one can show that, with the knowledge of `a1`, `a2`, and `m`, Alice can always infer the values of `x1` and `x2`, no matter what Bob's input values are [3]. Similarly, Bob can also infer the values of `x1` and `x2` from the knowledge of `b1`, `b2`, and `m`. Therefore, declassifying values

[1] Throughout this paper, we assume the *honest-but-curious* threat model (also called *semi-honest*), where parties always follow a prearranged protocol to its completion. When parties are acting honestly they do not, for instance, unexpectedly stop participating halfway through the protocol.

of x1 and x2 *explicitly* to Alice and Bob during the computation would not compromise privacy, since they can infer them anyway, and it turns out that doing so it enables the following, more efficient SMC protocol:

1. Alice and Bob compute a1 $\leq$ b1 using secure computation and share the output x1 (line 5).
2. Alice *locally* computes a3 (line 6).
3. Bob *locally* computes b3 (line 7).
4. Alice and Bob compute a3 $\leq$ b3 using secure computation and share the output x2 (line 8).
5. If x2 is true, Alice sends a3 to Bob as the final median, else if x2 is false, Bob sends Alice b3 as the final median (line 9 and 10).

Thus, in the optimized version, only the two comparisons (line 5 and 8), need to be done securely. Moreover, Alice and Bob do not learn anything more than they did in the unoptimized version. For median computation on a joint set with 64 elements, Kerschbaum [14] shows 30x performance improvement using this optimization.

This paper explores methods for inferring when and if variables like x1 and x2 in this example can be inferred by SMC participants, and thus may enable protocol optimizations like the one above. Specifically, we consider two related problems: *knowledge inference* and *constructive knowledge inference*. Both problems are specified by giving an unoptimized reference protocol for an SMC as a program that uses multiple parties' variables (as in Figure 1). From this reference protocol, a solution to the knowledge inference problem states which parties can learn which additional variables, if any, from a cooperative run of the protocol. We call a knowledge inference solution *constructive* if, in addition to correctly asserting that a party p knows a variable y, the solution also gives gives a means for party p to compute y from their private data and the final output. More generally, we say that the evidence of a party's knowledge is *constructive* if this evidence is a program that always correctly computes the value of variable that is claimed to be known (e.g., x1 or x2) from what will always be known by the party in question (e.g., a1, a2 and m).

Contributions. We make the following contributions:

- Within the context of secure multi-party computations, we formally define notions of *knowledge* (of a program variable y by a party p), and the problems of *knowledge inference* and *constructive knowledge inference*.
- We give a solution to the knowledge inference problem. We prove that our solution is sound (Theorem 5). Additionally, for the language that we consider, we also show that our solution is complete (Theorem 6).
- We give a solution to the constructive knowledge inference problem. We prove that our solution is sound (Theorem 7), and we characterize conditions under which it is also complete (Theorems 8 and 9).
- We implement our solutions and evaluate them experimentally.

Paper guide. Our paper is organized as follows. In the context of an example program, Section 2 informally presents our definitions of knowledge, knowledge inference and constructive knowledge inference. Section 3 presents formal definitions for the concepts introduced earlier, and explains our formal results in detail. In particular, Sections 3.2 and 3.3 present our algorithms for knowledge inference and constructive knowledge inference, respectively. Section 4 discusses alternative formulations to those of Section 3, and relates our approach to standard ones based on commonly-used notions of noninterference. Section 5 describes the implementation of our approach, and presents a preliminary empirical evaluation. Section 6 discussed related work and Section 7 concludes.

2. Overview

In this section, we present an overview of our knowledge inference approaches with the help of some examples.

In our setting, party p knows the (deterministic) program (call it S), his own input set I, and his output O.[2] We say party p can *infer* the value of local variable $y \in S$ if there exists a function F such that $y = F(I, O)$ in all runs of S. Another way of putting it is that no matter the values of p's inputs or those of other participants of the SMC, p can always compute y given knowledge of only his inputs and the final result. Our goal to find all those variables in S that p can infer. We can do this by either showing merely that the required function F exists, without saying what it is, or we can produce F directly, thus constituting a constructive proof. In this paper we present approaches to both tasks.

2.1 Knowledge inference

To show that an intermediate variable can be expressed as a function of one party's inputs and outputs, we can attempt to prove that given any pair of runs of S that agree on the valuations of variables in I and O (but may not agree on the input and output variables of other parties), the valuations of y on those two runs must also agree. In other words, y can be determined uniquely from I and O, and thus a function F exists such that $F(I, O) = y$. We can construct such a proof in two steps.

First we use a program analysis to produce a formula ϕ_{post} that soundly approximates the final state of the program S (that is, the final values of all program variables) for all possible program runs. So that the meaning of a variable y mentioned in ϕ_{post} is unambiguous, we assume that a variable is assigned at most once during a program run.

[2] Some SMCs may have different outputs for different parties; in the median example, there is a single output O = m known to both parties.

```
## a1 < a2, b1 < b2, distinct(a1, a2, b1, b2)
int m = median(a1, a2, b1, b2);

## a1'<a2', b1'<b2', distinct(a1', a2', b1', b2')
int m' = median'(a1', a2', b1', b2');
```

Figure 2. Median computation composed with itself.

One program analysis we might use to produce ϕ_{post} is symbolic execution [15]. Each feasible program path is characterized by a *path condition*, which is a set of predicates relating the program variables. Each feasible path condition φ_i can be combined to provide a complete description of the program's behavior: $\phi_{post} \stackrel{\text{def}}{=} \bigvee_i \varphi_i$. For the median program of Figure 1, there are four possible paths, having the following path conditions:

$$\begin{aligned}
\varphi_1 &\stackrel{\text{def}}{=} \texttt{a1} < \texttt{b1} \wedge \texttt{x1} = \mathsf{true} \wedge \texttt{a3} = \texttt{a2} \wedge \texttt{b3} = \texttt{b1} \wedge \\
&\quad \texttt{a3} < \texttt{b3} \wedge \texttt{x2} = \mathsf{true} \wedge \texttt{m} = \texttt{a3} \wedge \phi_{pre} \\
\varphi_2 &\stackrel{\text{def}}{=} \texttt{a1} < \texttt{b1} \wedge \texttt{x1} = \mathsf{true} \wedge \texttt{a3} = \texttt{a2} \wedge \texttt{b3} = \texttt{b1} \wedge \\
&\quad \texttt{a3} \geq \texttt{b3} \wedge \texttt{x2} = \mathsf{false} \wedge \texttt{m} = \texttt{b3} \wedge \phi_{pre} \\
\varphi_3 &\stackrel{\text{def}}{=} \texttt{a1} \geq \texttt{b1} \wedge \texttt{x1} = \mathsf{false} \wedge \texttt{a3} = \texttt{a1} \wedge \texttt{b3} = \texttt{b2} \wedge \\
&\quad \texttt{a3} < \texttt{b3} \wedge \texttt{x2} = \mathsf{true} \wedge \texttt{m} = \texttt{a3} \wedge \phi_{pre} \\
\varphi_4 &\stackrel{\text{def}}{=} \texttt{a1} \geq \texttt{b1} \wedge \texttt{x1} = \mathsf{false} \wedge \texttt{a3} = \texttt{a1} \wedge \texttt{b3} = \texttt{b2} \wedge \\
&\quad \texttt{a3} \geq \texttt{b3} \wedge \texttt{x2} = \mathsf{false} \wedge \texttt{m} = \texttt{b3} \wedge \phi_{pre}
\end{aligned}$$

Consider the first path condition φ_1. Conceptually, it describes all program states in which the **then** branch of both conditionals (lines 6 and 9). The remaining three paths constitute the other three possible branching combinations. Together, the four paths fully describe the possible feasible states at the end of the median function, Namely, any valuation of the variables reaching the end must satisfy this formula. Note that each path also requires ϕ_{pre}. This formula defines the publicly-known constraints on all inputs; in the case of the median program we have $\phi_{pre} \stackrel{\text{def}}{=} \texttt{a1} < \texttt{a2} \wedge \texttt{b1} < \texttt{b2} \wedge \texttt{a1} \neq \texttt{b1} \wedge \texttt{a1} \neq \texttt{b2} \wedge \texttt{a2} \neq \texttt{b1} \wedge \texttt{a2} \neq \texttt{b2}$.

The next step is to prove that any two runs of the program S that agree on variables known to p will also agree on the value of y. This statement is a 2-safety property [7], and we can prove it using a technique called self-composition [4]. The idea is to reduce this two-run condition on program S to a condition on a single run of a *self-composed program* S_c, which is the sequential composition of S with itself, with the second copy of S's variables renamed, e.g., so that x is renamed to x'. Given the formula ϕ^{sc}_{post} for this self-composed program, we can ask whether, under the assumption that the normal and primed versions of p-visible variables are equal, that the normal and primed version of y is also equal.

As an example, Figure 2 shows self composition of the median function of Figure 1. We write $\texttt{median}'$ for the function `median` but with the local variables renamed to $\texttt{x1}', \texttt{x2}', ...$ The self-composed program effectively runs median twice, on two separate spaces of variables. We can express the question of knowledge inference as a question on the relationship between the two copies of the variables. Namely, Alice can infer `x1` if and only if for every feasible final state of the composed program, when the two copies of $\texttt{a1}, \texttt{a2}, \texttt{m}$ agree on their values then the copies of `x1` agree on their value. More formally we need to check the validity of the following formula for any feasible final state.

$$\phi^{sc}_{post} \wedge (\texttt{a1} = \texttt{a1'} \wedge \texttt{a2} = \texttt{a2'} \wedge \texttt{m} = \texttt{m'}) \Rightarrow (\texttt{x1} = \texttt{x1'})$$

Here, the formula ϕ^{sc}_{post} will involve sixteen path conditions (self-composition squares the number of paths). For example, among them will be:

$$\begin{aligned}
\varphi^{sc}_1 &\stackrel{\text{def}}{=} \texttt{a1} < \texttt{b1} \wedge \texttt{x1} = \mathsf{true} \wedge \texttt{a3} = \texttt{a2} \wedge \texttt{b3} = \texttt{b1} \wedge \\
&\quad \texttt{a3} < \texttt{b3} \wedge \texttt{x2} = \mathsf{true} \wedge \texttt{m} = \texttt{a3} \wedge \phi_{pre} \wedge \\
&\quad \texttt{a1'} < \texttt{b1'} \wedge \texttt{x1'} = \mathsf{true} \wedge \texttt{a3'} = \texttt{a2'} \wedge \texttt{b3'} = \texttt{b1'} \wedge \\
&\quad \texttt{a3'} < \texttt{b3'} \wedge \texttt{x2'} = \mathsf{true} \wedge \texttt{m'} = \texttt{a3'} \wedge \phi'_{pre}
\end{aligned}$$

The formula ϕ^{sc}_1 is actually the conjunction of ϕ_1 with a version of ϕ_1 that has all its variables renamed to the primed versions. We can think of the entire post condition $\phi^{sc}_{post} = \phi^{sc}_1 \wedge ... \wedge \phi^{sc}_{16}$ as the conjunction of the post condition ϕ_{post} with its primed version. Thus self-composition serves as conceptual aid here; the actual construction does not need to be performed to derive ϕ^{sc} as it can be formulated as two runs of the original program using different variable names.

Being a quantifier-free formula in the theory of integer linear arithmetic, the final formula poses no problem for an SMT solver such as Z3 [2], which can indeed verify its validity. Additionally, the same can be said for Alice's knowledge of `x2` and `a3`, and Bob's knowledge of `x1`, `x2` and `b3`.

The knowledge inference question as we have expressed it here bears a close a resemblance to deciding the property of *delimited release* [20]. We explore this connection in more detail in Section 4.

2.2 Constructive Knowledge Inference

The technique just described can establish that there exists some function F such that $y = F(I, O)$, where y is an intermediate variable in S, and I and O are variables known to party p. However, this technique cannot say what F actually is. To construct F we can leverage ideas from template-based program verification.

Program verification generally aims at inferring invariants in a program that are strong enough to verify some assertions of interest. Template-based program verification [11, 21] requires programmers to specify the structure of these invariants in the form of a template. The algorithm then generates verification constraints for the assertions, to be solved by an SMT solver (e.g. Z3 [2]). A solution to the constraints yields a valid proof of the correctness of the assertions as well as a solution to the template unknowns. Gulwani et. al. [11] present constraint-based verification techniques over the abstraction of linear arithmetic and Srivastava et. al. [21] present these techniques over the abstraction of predicates.

To infer p's knowledge of a variable y, our algorithm tries to infer a formula ϕ s.t. (a) at the end of the program $y = \phi$, and (b) ϕ only mentions the input and output variables known to p. If we add an assertion $y = \phi$ at the end of the program, and provide a template structure for ϕ (limited to formulae over variables known to p) this becomes a template-driven verification problem where the assertion and the invariant are the same. A successful verification of the assertion $y = \phi$ establishes p's knowledge of y, and also the solution for ϕ yields a formula for y in terms of input and output variables of p.

The template structure for this problem is defined as follows. Suppose y is a boolean variable. Then the template for ϕ requires it to be in *disjunctive normal form* (DNF) such that there are exactly d disjuncts each consisting of c conjuncts, with each conjunct drawn from a set of predicates Q. This set contains predicates over linear expressions involving I and O. In the median example, for Alice, one choice of Q is $\{v_1 \odot v_2 \mid v_1, v_2 \in \{\mathtt{a1}, \mathtt{a2}, \mathtt{m}\}, \odot \in \{>, \geq, <, \leq, =, \neq\}\}$. For Bob, similar Q would be $\{v_1 \odot v_2 \mid v_1, v_2 \in \{\mathtt{b1}, \mathtt{b2}, \mathtt{m}\}, \odot \in \{>, \geq, <, \leq, =, \neq\}\}$.

Our algorithm searches for a ϕ conforming to the prescribed template. For example, if $(c = 2, d = 2)$, then the search space for ϕ is all the boolean formulae $(q_1 \wedge q_2) \vee (q_3 \wedge q_4), q_1, q_2, q_3, q_4 \in Q$. We denote this search space for formulae as $\mathsf{DNF}(c, d, Q)$. A naive search algorithm would make $O(|Q|^{cd})$ queries to the SMT solver, one for every possible formula in $\mathsf{DNF}(c, d, Q)$. This algorithm is complete in the sense that if there exists a solution for ϕ in $\mathsf{DNF}(c, d, Q)$ then the naive search algorithm finds it. Our algorithm, on the other hand, makes $O(|Q|^c + |Q|^d)$ queries to the SMT solver, and still guarantees completeness, provided the existence of solution in $\mathsf{DNF}(c, d, Q)$.

Consider variable `x1` from the median example. With Q_{Alice} and Q_{Bob} as above, and $(c = 1, d = 1)$, we are able to establish knowledge of `x1` for both Alice and Bob as: $\mathtt{x1} = \mathtt{m} > \mathtt{a1}$ for Alice, and $\mathtt{x1} = \mathtt{m} \leq \mathtt{b1}$ for Bob. Interestingly, $(c = 1, d = 1)$ is insufficient to discover invariants describing Alice's and Bob's knowledge of `x2`. With $(c = 1, d = 2)$, we are able to establish Alice's knowledge of `x2` as $\mathtt{x2} = (\mathtt{m} = \mathtt{a1} \vee \mathtt{m} = \mathtt{a2})$. And with $(c = 2, d = 1)$, we are able to establish Bob's knowledge of `x2` as $\mathtt{x2} = (\mathtt{m} \neq \mathtt{b1} \wedge \mathtt{m} \neq \mathtt{b2})$. In general, starting with $(c = 1, d = 1)$, we can increment (c, d) in steps until either we find a solution or we leave x as being unknown to p.

We can also infer formulae for integer variables. In this case we use a different template structure, and leverage ideas from Gulwani et. al. [11]. We discuss the algorithm further in the next section.

Value	v	$::=$	$n \mid \mathsf{true} \mid \mathsf{false}$
Exprn./Formula	e, ϕ	$::=$	$v \mid x \mid e_1 \odot e_2$
Binary operator	$\odot$	$\in$	$\{\leq, \geq, >, <, =, \neq\}$
		$\cup$	$\{\wedge, \vee, \neg, \Rightarrow\} \cup \{+, -\}$
Statement	S	$::=$	$x := e \mid S_1; S_2 \mid \mathsf{skip}$
		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$

Figure 3. Syntax.

3. Formal Development

In this section, we formally describe our knowledge inference algorithm. We first give the language syntax, operational semantics, and formal definition of *knowledge* in Section 3.1. We then present the inference algorithm in Section 3.2, and the constructive inference algorithm in Section 3.3.

3.1 Language Syntax

Let parties $p_1, \ldots, p_n$ want to compute a secure computation S whose syntax is given in Figure 3. The language is standard aside from the omission of a looping construct; this makes sense in our setting since most SMC methods forbid dynamic looping (rather, they require a static loop unrolling). Our methods support loops as well, but we elide them nevertheless to keep the formalization simpler. We also assume, for simplicity, that each program path is in single assignment form, i.e. in an execution of a program, every variable is assigned at most once.

The semantics of computations is given in Figure 4. The judgments have the form $\langle \sigma, S \rangle \Downarrow \sigma'$, meaning, statement S executed in state σ results in new state σ'. States σ are maps from variables x to values v; we write $\sigma[x]$ to look up x in σ, and we write $\sigma[x \mapsto v]$ to define a map identical to σ except that x maps to v. The figure also defines an auxiliary judgment for expressions having the form $\langle \sigma, e \rangle \Downarrow v$, meaning, expression e evaluated in state σ results in value v. The rules are standard, with one exception. The rule (E-ASSIGN) checks that $x \notin \sigma$ to enforce single assignment form for the current program path. When an expression is viewed as a formula ϕ, we write $\sigma \models \phi$ to mean that in σ the formula ϕ evaluates to true. We also write predicate to mean a boolean valued formula.

Let V be a set of variables. We define two states as being *equivalent* on a set of variables as follows:

Definition 1 (Equivalence of States)**.** Two states, σ_1 and σ_2, are equivalent on a set of variables V, written as $\sigma_1 \stackrel{V}{\equiv} \sigma_2$, iff $\forall x \in V$, $\sigma_1[x] = \sigma_2[x]$.

Let ϕ_{pre} denote the precondition for a secure computation program S. It represents the assumptions that S makes about

$$\text{(E-VAR)}\ \frac{}{\langle \sigma, x\rangle \Downarrow \sigma[x]} \qquad \text{(E-VAL)}\ \frac{}{\langle \sigma, v\rangle \Downarrow v} \qquad \text{(E-BINOP)}\ \frac{\langle \sigma, e_1\rangle \Downarrow v_1 \quad \langle \sigma, e_2\rangle \Downarrow v_2}{\langle \sigma, e_1 \odot e_2\rangle \Downarrow v_1 \odot v_2}$$

$$\text{(E-ASSIGN)}\ \frac{x \notin \sigma \qquad \langle \sigma, e\rangle \Downarrow v}{\langle \sigma, x := e\rangle \Downarrow \sigma[x \mapsto v]} \qquad \text{(E-SEQ)}\ \frac{\langle \sigma, S_1\rangle \Downarrow \sigma' \quad \langle \sigma', S_2\rangle \Downarrow \sigma''}{\langle \sigma, S_1; S_2\rangle \Downarrow \sigma''}$$

$$\text{(E-IFTRUE)}\ \frac{\langle \sigma, e\rangle \Downarrow \mathsf{true} \quad \langle \sigma, S_1\rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Downarrow \sigma'} \qquad \text{(E-IFFALSE)}\ \frac{\langle \sigma, e\rangle \Downarrow \mathsf{false} \quad \langle \sigma, S_2\rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Downarrow \sigma'}$$

$$\text{(E-SKIP)}\ \frac{}{\langle \sigma, \mathsf{skip}\rangle \Downarrow \sigma} \qquad (\phi\text{-VALID})\ \frac{\langle \sigma, \phi\rangle \Downarrow \mathsf{true}}{\sigma \models \phi}$$

Figure 4. Semantics.

parties' inputs. In the median example from Figure 1, $\phi_{pre} = \mathtt{a1} < \mathtt{a2} \wedge \mathtt{b1} < \mathtt{b2} \wedge \mathtt{a1} \neq \mathtt{b1} \wedge \mathtt{a1} \neq \mathtt{b2} \wedge \mathtt{a2} \neq \mathtt{b1} \wedge \mathtt{a2} \neq \mathtt{b2}$. We are interested in executions $\langle \sigma, S\rangle \Downarrow \sigma'$ when $\sigma \models \phi_{pre}$.

We now define *knowledge* of a variable y to a party p in S, written as $\mathfrak{K}(S, p, y)$. Informally, y is known to p, if, whenever two final states of S are equivalent on the set of input and output variables of p, they are also equivalent on $\{y\}$.

Definition 2. [Knowledge of a Variable] Let S be a secure computation program with precondition ϕ_{pre}. For a party p in the computation, let I be the set of input variables of p, and O be the set of output variables of p. Then, a variable y in S is *known* to p, written as $\mathfrak{K}(S, p, y)$, if for all initial states σ_1, σ_2 s.t. $\sigma_1 \models \phi_{pre}$, $\sigma_2 \models \phi_{pre}$, and $\sigma_1 \overset{I}{\equiv} \sigma_2$, whenever $\langle \sigma_1, S\rangle \Downarrow \sigma_1'$ and $\langle \sigma_2, S\rangle \Downarrow \sigma_2'$ s.t. $\sigma_1' \overset{O}{\equiv} \sigma_2'$, we have $\sigma_1' \overset{\{y\}}{\equiv} \sigma_2'$.

The definition models the 2-safety property discussed in the Section 2. It says that the value of y can be uniquely determined from the knowledge of input and output variables of p, independent of the inputs of other parties in the computation. We now give the formal description of knowledge inference algorithm.

3.2 Knowledge Inference

The problem of knowledge inference is as follows. For a secure computation program S, we want to know whether a party p *knows* a program variable y according to Definition 2. We present our knowledge inference algorithm in Figure 7, but before that we give some auxiliary definitions.

Definition 3 (Validity of a Predicate)**.** A predicate ϕ is valid at the end of a program S with precondition ϕ_{pre}, if $\forall \sigma$ s.t. $\sigma \models \phi_{pre}$, $\langle \sigma, S\rangle \Downarrow \sigma'$, we have $\sigma' \models \phi$.

$$\begin{array}{lcl}
\varsigma(\mathsf{skip}, \phi) & = & \phi \\
\varsigma(\mathsf{x := e}, \phi) & = & \phi \wedge (x = e) \\
\varsigma(\mathsf{S_1; S_2}, \phi) & = & \varsigma(S_2, \varsigma(S_1, \phi)) \\
\varsigma(\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \phi) & = & (e \wedge \varsigma(S_1, \phi)) \vee \\
 & & (\neg e \wedge \varsigma(S_2, \phi))
\end{array}$$

Figure 5. Postcondition of a predicate ϕ w.r.t. statement S.

$$\text{(T-VAR1)}\ \frac{x \in \theta}{\langle \theta, x\rangle \rightsquigarrow \langle \theta, \theta[x]\rangle} \qquad \text{(T-VAR2)}\ \frac{x \notin \theta \quad x' \text{ is fresh}}{\langle \theta, x\rangle \rightsquigarrow \langle \theta[x \mapsto x'], x'\rangle}$$

$$\text{(T-BINOP)}\ \frac{\langle \theta, \phi_1\rangle \rightsquigarrow \langle \theta', \phi_1'\rangle \quad \langle \theta', \phi_2\rangle \rightsquigarrow \langle \theta'', \phi_2'\rangle}{\langle \theta, \phi_1 \odot \phi_2\rangle \rightsquigarrow \langle \theta'', \phi_1' \odot \phi_2'\rangle} \qquad \text{(T-VAL)}\ \frac{}{\langle \theta, v\rangle \rightsquigarrow \langle \theta, v\rangle}$$

Figure 6. Variable renaming translation for a predicate.

We define the postcondition of a predicate ϕ w.r.t. statement S, written as $\varsigma(S, \phi)$, in Figure 5. The following theorem states the properties of $\varsigma(S, \phi)$.

Theorem 4. *[Soundness and Completeness of Postcondition] For a program S with precondition ϕ_{pre}, $\varsigma(S, \phi_{pre})$ is valid at the end of program S (Soundness). Moreover, for any other predicate ϕ s.t. ϕ is valid at the end of S, $\varsigma(S, \phi_{pre}) \Rightarrow \phi$ (Completeness).*

Proof. Soundness – Structural induction on S. Completeness – Structural induction on S and using following lemma for each case. Let $\sigma \models \varsigma(S, \phi_{pre})$. Then, $\exists \sigma'$ s.t. $\sigma' \models \phi_{pre}$, $\langle \sigma', S\rangle \Downarrow \sigma$, and $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) - \mathsf{Def}(S)$, where $\mathsf{Def}(S)$ is the set of variables defined by S. ☐

The theorem depends on the program paths being in single assignment form. Specifically, the postcondition rule for assignment statement assumes that x does not occur in ϕ.

We now define a variable renaming translation on predicates. The idea is to replace every variable in the predicate with a *copy* of the variable. Let θ be a mapping from variables to variables. The translation judgment is shown in Figure 6. We define similar translation judgments for statements and states and refer to them in the theorem proofs later on, however we do not show them here for lack of space.

Our algorithm is parameterized by an SMT solver (e.g. Z3 [2], STP [9]), that we denote as `alg`. We use `alg` to determine whether a given predicate is a tautology (always true). We write $\vdash_{\mathtt{alg}} \phi$ as the query to `alg` for predicate ϕ.

The knowledge inference algorithm is shown in Figure 7. It takes as input the secure computation program S and its precondition ϕ_{pre}. For each party p and program variable y, it outputs whether p knows y or not.

The algorithm first computes the postcondition of ϕ_{pre} w.r.t. S (ϕ_{post}). It then performs variable translation on ϕ_{post} to generate ϕ_{post}'. Essentially ϕ_{post} and ϕ_{post}' model two

```
1  InferKnowledge(S, φ_pre)
2    for each party p
3      let I be the set of p's input variables.
4      let O be the set of p's output variables.
5      φ_post := ς(S, φ_pre);
6      ⟨ε, φ_post⟩ ⇝ ⟨θ, φ'_post⟩.
7      φ_k := ⋀_{x∈I∪O} (x = θ[x]);
8      for each program variable y
9        φ := (φ_post ∧ φ'_post ∧ φ_k) ⇒ (y = θ[y]);
10       if(⊢_alg φ)
11         output y is known to p.
12       else
13         output y is not known to p.
```

Figure 7. Knowledge inference algorithm. ϕ_{pre} is the precondition of S. The algorithm first generates postconditions for two *different* runs of S (ϕ_{post} and ϕ'_{post}). To establish p's knowledge of a program variable y, it then tries to prove, using `alg`, that whenever these two runs are equivalent on p's input and output variables, they are also equivalent on y.

different runs of the program. ϕ_k then asserts that $\forall x \in I \cup O$, x has same value across these two runs. Under these assumptions, if a program variable y also has same value across these two runs, the variable y is known to p.

The soundness theorem of our algorithm is as follows:

Theorem 5 (Soundness of Knowledge Inference)**.** *Let S be a secure computation program with the precondition ϕ_{pre}. If InferKnowledge(S, ϕ_{pre}) outputs variable y is known to party p, then $\mathfrak{K}(S, p, y)$.*

Proof. We want to prove that for two states σ and σ' s.t. $\sigma \overset{I\cup O}{\equiv} \sigma'$, $\sigma[y] = \sigma[y']$ (see Definition 2). If we translate σ according to θ to yield σ' ($\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \epsilon$), then we can see that $\sigma \cup \sigma' \models \phi_{post}$, $\sigma \cup \sigma' \models \phi'_{post}$ (soundness of postcondition), and $\sigma \cup \sigma' \models \phi_k$ (using above equivalence). Thus, it follows from line 9 in Figure 7 that $\sigma \cup \sigma' \models (y = \theta[y])$. □

Moreover, we can also state a completeness theorem.

Theorem 6 (Completeness of Knowledge Inference)**.** *Let S be a secure computation program with precondition ϕ_{pre}. For a program variable y and party p, if $\mathfrak{K}(S, p, y)$, then InferKnowledge(S, ϕ_{pre}) outputs variable y is known to party p.*

Proof. For a program S, let S' be the translation of S. Then, we can see that $\phi_{post} \wedge \phi'_{post} \wedge \phi_k = \varsigma(S; S', \phi_{pre} \wedge \phi'_{pre} \wedge \phi_k)$. By completeness of postcondition, if $y = \theta[y]$ is valid at the end of $S; S'$, line 9 in Figure 7 must be true. □

```
ConstructKnowledgeB(S, φ_pre)
  for each party p
    construct the predicate set Q.
    for each boolean program variable y
      c = 1; d = 1;
      do
        φ := CFormula(y, c, d, Q);
        increment (c, d) in lockstep.
      while (φ is failure and c < c_max, d < d_max);
      if(φ = failure)
        output y is not known to p.
      else
        output y is known to p by φ.
```

Figure 8. Constructive knowledge inference for boolean variables. For each party p and each boolean program variable y, starting with $(c = 1, d = 1)$, it calls `CFormula` (Figure 9) to construct a formula for y in $\mathsf{DNF}(c, d, Q)$ template.

3.3 Constructive Knowledge Inference

The knowledge inference algorithm from Figure 7 establishes whether p knows y or not, however it does not give a formula for y in terms of p's input and output variables. In this section, we present constructive knowledge inference algorithms, that output such a formula.

Constructive knowledge inference for boolean variables. Define the verification condition of a predicate ϕ w.r.t. a statement S with precondition ϕ_{pre}, $\mathtt{VC}(S, \phi_{pre}, \phi)$, as $\varsigma(S, \phi_{pre}) \Rightarrow \phi$. Then, if $\vdash_{\mathtt{alg}} \mathtt{VC}(S, \phi_{pre}, \phi)$, the predicate ϕ is valid at the end of S.

Recall that to construct knowledge of variable y for a party p, we want to infer a formula ϕ, s.t. at the end of the program $x = \phi$ holds. For boolean variables, the search space for ϕ is $\mathsf{DNF}(c, d, Q)$, where Q is a set of predicates constructed from input and output variables of p.

The algorithm for boolean variables is shown in Figure 8. For each party p, it first constructs a predicate set Q. As mentioned earlier, this can either be provided as input by the programmer, or it can be mined from the expressions appearing in the program. For each boolean program variable y, starting with $(c = 1, d = 1)$ and incrementing (c, d) in lockstep until (c_{max}, d_{max}), it tries to find ϕ. It uses an auxiliary routine `CFormula`, defined in Figure 9.

Figure 9 consists of the subroutine `CFormula` and two other subroutines, that it invokes, `CFormulaL` and `CFormulaR`. We divide the problem of constructing ϕ into subproblems of constructing ϕ_L and ϕ_R s.t. (a) ϕ_L and ϕ_R consist only of predicates from Q, and (b) at the end of the program, $\phi_L \Rightarrow y$, $y \Rightarrow \phi_R$, and $\phi_R \Rightarrow \phi_L$ hold. Then, we have $\phi = \phi_R$. `CFormulaL` constructs ϕ_L and `CFormulaR` constructs ϕ_R.

Construction of ϕ_L. To construct ϕ_L (`CFormulaL` in Figure 9), we perform breadth first search on the lattice of sub-

```
CFormula($y, c, d, Q$) ## construct $\phi$ s.t. $y \Leftrightarrow \phi$
  let $\mathcal{L}$ be the lattice ($2^Q, \Rightarrow, \top = \{\}, \bot = Q$).
  $\phi_L :=$ CFormulaL($y, c, \mathcal{L}$); $\phi_R :=$ CFormulaR($y, d, Q$);
  if($\phi_L$ = failure || $\phi_R$ = failure)
      return failure;
  $\phi :=$ VC($S, \phi_{pre}, \phi_R \Rightarrow \phi_L$);
  if($\vdash_{\texttt{alg}} \phi$)
    return $\phi_R$;
  else
    return failure;

CFormulaR($y, d, Q$) ## construct $\phi_R$ s.t. $y \Rightarrow \phi_R$
  $\mathcal{N} := \{\}$; ## set of tuples that satisfy $y \Rightarrow \phi_R$
  for all $(q_1, \ldots, q_d) \in (Q \times_1 Q \cdots \times_{d-1} Q)$
    $\phi :=$ VC($S, \phi_{pre}, y \Rightarrow \bigvee_{i=1}^{d} q_i$);
    if($\vdash_{\texttt{alg}} \phi$)
      $\mathcal{N} := \mathcal{N} \cup \{(q_1, \ldots, q_d)\}$;
  if($\mathcal{N} = \{\}$)
    return failure;
  else
    return $\bigwedge_{(q_1,\ldots,q_d)\in\mathcal{N}} (\bigvee_{i=1}^{d} q_i)$;
```

```
CFormulaL($y, c, \mathcal{L}$) ## construct $\phi_L$ s.t. $\phi_L \Rightarrow y$
  $\mathcal{N} := \{\}$; ## set of lattice nodes that satisfy $\phi_L \Rightarrow y$
  visit lattice $\mathcal{L}$ nodes in BFS order,
    when node $N$ is visited, do
      if($N = \{\}$)
        $\phi_N :=$ true;
      else
        let $N$ be $\{q_1, \ldots, q_n\}$.
        $\phi_N := \bigwedge_{i=1}^{n} q_i$;
      $\phi :=$ VC($S, \phi_{pre}, \phi_N \Rightarrow y$);
      if($\vdash_{\texttt{alg}} \phi$)
        $\mathcal{N} := \mathcal{N} \cup \{N\}$;
        truncate sublattice rooted at $N$ from BFS.
      else
        for each child $M$ of $N$ in $\mathcal{L}$
          if($|M| \leq c$ && $M$ is unvisited)
            add $M$ to BFS worklist.
  if($\mathcal{N} = \{\}$)
    return failure;
  else
    return $\bigvee_{\{q_1,\ldots,q_n\}\in\mathcal{N}} (\bigwedge_{i=1}^{n} q_i)$;
```

Figure 9. The routine `CFormula` constructs a formula for y in $\mathsf{DNF}(c, d, Q)$ template. It calls the subroutine `CFormulaL` to construct ϕ_L s.t. $\phi_L \Rightarrow y$, and subroutine `CFormulaR` to construct ϕ_R s.t. $y \Rightarrow \phi_R$. Finally, it checks that $\phi_R \Rightarrow \phi_L$, and if so, returns ϕ_R as the solution for y.

sets of Q ordered by implication (i.e. $M \sqsubseteq N, M, N \in 2^Q$, iff $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} p) \Rightarrow (\bigwedge_{q' \in N} q')$) with $\top = \{\}$ and $\bot = Q$, and collect all nodes of the lattice that form a *solution* to $\phi_L \Rightarrow y$. A node N in the lattice is a solution to $\phi_L \Rightarrow y$ if $\vdash_{\texttt{alg}}$ $\texttt{VC}(S, \phi_{pre}, (\bigwedge_{q \in N} q) \Rightarrow y)$. When we find a node N that is a solution, we delete the subtree rooted at N from the lattice, since any node in the subtree is a "weaker" solution than N (i.e. for any node M in the subtree under N, we have $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} q) \Rightarrow (\bigwedge_{q' \in N} q')$, and since $\vdash_{\texttt{alg}} (\bigwedge_{q' \in N} q') \Rightarrow y$, we already have $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} q) \Rightarrow y$). Moreover, we also prune any subtree rooted at a node (including the node itself) whose size is greater than c (since the current search space is $\mathsf{DNF}(c, d, Q)$, we need not consider lattice nodes with more than c elements). Let $\mathcal{N}$ be the set of lattice nodes that are found as solutions. We assign $\phi_L = \bigvee_{N \in \mathcal{N}} (\bigwedge_{q \in N} q)$. If $\mathcal{N} = \{\}$, the algorithm fails to infer p's knowledge of y (under input values of c and d). Construction of ϕ_L makes $O(|Q|^c)$ queries to the SMT solver.

Construction of ϕ_R***.*** To construct ϕ_R (`CFormulaR` in Figure 9), we consider all possible $(q_1, \ldots, q_d) \in (Q \times_1 Q \cdots \times_{d-1} Q)$, and collect all such tuples that form a *solution* to $y \Rightarrow \phi_R$. $(q_1, \ldots, q_d)$ is a *solution* to $y \Rightarrow \phi_R$ if $\vdash_{\texttt{pre}}$ $\texttt{VC}(S, \phi_{pre}, y \Rightarrow \bigvee_{i=1}^{d} q_i)$. Let $\mathcal{N}$ be the set of such solutions. Then, we assign $\phi_R = \bigwedge_{(q_1,\ldots,q_d)\in\mathcal{N}} (\bigvee_{i=1}^{d} q_i)$. If $\mathcal{N} = \{\}$, the algorithm fails to infer P's knowledge of y (under input values of c and d). Construction of ϕ_R makes $O(|Q|^d)$ queries to the SMT solver.

Construction of ϕ***.*** We now check that $\phi_R \Rightarrow \phi_L$ is valid at the end of the program using the formulae for ϕ_L and ϕ_R constructed above (`CFormula` in Figure 9). If it is, y is known to p using the formula ϕ_2, otherwise our algorithm returns y is not known to p (under input values of c and d).

Constructive knowledge inference for integer variables. For integer variables in the program S, constructive knowledge inference algorithm is shown in Figure 10. To verify ϕ on line 6, we use the algorithm given by Gulwani et. al. [11]. Their algorithm uses Farka's lemma to convert ϕ into SAT solver constraints, the solution of which returns a solution for the template unknowns a_i s.t. ϕ holds true at the end of S, and thus, $y = \sum_{i=1}^{n} a_i x_i$.

We state soundness theorems for constructive knowledge inference algorithms as follows:

Theorem 7 (Soundness of Constructive Knowledge Inference)**.** *Let S be a secure computation program with pre-*

1 `ConstructKnowledgeI`(S, ϕ_{pre})
2   `for each party` p
3   `let` $\{x_i\}^{i \in 1 \ldots n}$ `be input and output variables of` p.
4   `for each integer program variable` y
5   `let` $a_i, i \in 1 \ldots n$ `be` n `integer unknowns.`
6   ϕ `:=` $-y + \sum_{i=1}^{n} a_i x_i \geq 0 \ \wedge y + \sum_{i=1}^{n} - a_i x_i \geq 0$;
7   `verify` ϕ `at the end of` S.
8   `if verification fails`
9   `output` y `is not known to` p.
10   `else`
11   `output` y `is known to` p `by` $\sum_{i=1}^{n} a_i x_i$.

Figure 10. Constructive knowledge inference for integer variables. For each integer variable y, it tries to find a linear arithmetic formula for y in terms of input and output variables of p. To verify ϕ on line 6, it uses the algorithm by Gulwani et. al. [11].

condition ϕ_{pre}. If `ConstructKnowledgeB`(S, ϕ_{pre}) (Figure 8) and `ConstructKnowledgeI`(S, ϕ_{pre}) (Figure 10) output variable y is known to party p, then $\mathfrak{K}(S, p, y)$.

Proof. If the algorithms infer $y = \phi$ for a party p, then $y = \phi$ is valid at the end of S. Moreover, since only variables in ϕ are variables from $I \cup O$ (input and output variables of p), p knows y by Definition 2. $\square$

Moreover, constructive knowledge inference algorithms are also complete, provided a solution exists in the template form they consider.

Theorem 8 (Completeness of Constructive Knowledge Inference for Boolean Variables)**.** *Let S be a secure computation program. For a party p, let Q be a set of predicates, where the only variables appearing in each predicate in Q are input and output variables of p. Let y be a boolean program variable in S. If $\exists \phi$ s.t. $x = \phi$ at the end of S, and ϕ is in $\mathsf{DNF}(c, d, Q)$ form, for some values of (c, d), then `CFormula`(y, c, d, Q) returns a solution (and not `failure`).*

Proof. We give an outline for $(c = 2, d = 2)$, the proof for general case follows similarly. Let $y = \phi$ at the end of S s.t. ϕ is in $\mathsf{DNF}(c, d, Q)$ form. Then, for some $q_1, q_2, q_3, q_4 \in Q$, $y = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, equivalently, $y = (q_1 \vee q_3) \wedge (q_1 \vee q_4) \wedge (q_2 \vee q_3) \wedge (q_2 \vee q_4)$. Since `CFormulaR` considers all elements in $Q \times Q$, it would construct $\phi_R = (q_1 \vee q_3) \wedge (q_1 \vee q_4) \wedge (q_2 \vee q_3) \wedge (q_2 \vee q_4) \wedge \phi'$, for some ϕ' (possibly just true). On the other hand, since `CFormulaL` considers all lattice nodes up to size c, it would construct $\phi_L = (q_1 \wedge q_2) \vee (q_3 \wedge q_4) \vee \phi''$ for some ϕ'' (possibly just false). We can see that $\phi_R \Rightarrow \phi_L$, and hence `CFormula` returns ϕ_R. $\square$

The following theorem of completeness for integer variables follows from the completeness of the algorithm by Gulwani et. al. [11][3].

Theorem 9 (Completeness of Constructive Knowledge Inference for Integer Variables)**.** *Let S be a secure computation program. Let y be an integer variable in S. For a party p, let $\{x_i\}^{i \in 1 \ldots n}$ be the set of input and output variables of p. If $\exists a_i, i \in 1 \ldots n$ s.t. $y = \sum_{i=1}^{n} a_i x_i$ at the end of S, then `ConstructKnowledgeI`(S, ϕ_{pre}) (Figure 10) outputs y is known to p.*

Proof. Follows from the completeness of [11]. $\square$

4. Discussion

This section considers some aspects of our approach, including the relationship of knowledge inference to the property of *delimited release* [20], the effect of using a different program analysis to determine a program's final states, the possible use of type-based information flow analysis for knowledge inference, and finally the application of knowledge inference to allowing SMC computations with loops.

Relating knowledge inference to noninterference. As mentioned in Section 2.1, the knowledge inference problem bears some resemblance to the problem of proving noninterference, as evidenced by the similarity of our use of self-composition with its previous use in proving noninterference [4]. More precisely, knowledge inference is closely related Sabelfeld and Myers' *delimited release* [20] property. Next we define delimited release, and then show how a method for proving a program satisfies delimited release can be applied to knowledge inference.

In the setting of normal delimited release, we suppose there exists a *security labeling* Γ, which maps each program variable in S to one of two security labels, L (low) and H (high). We say that memories σ_1 and σ_2 are *low-equivalent*, written $\sigma_1 \sim_\Gamma \sigma_2$, if $\sigma_1(x) = \sigma_2(x)$ for all variables x such that $\Gamma(x) = L$. We also suppose that the program S may contain expressions `declassify(`e`)`, which signal that e's security label should be considered L, even if its contents may otherwise suggest its label should be H. (In an SMC, we can think of the output as being declassified; e.g., in Figure 1, we would change line 10 to be `return declassify(m)`.) We say that S enjoys *delimited release* with respect to Γ *iff* for all memories $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ such that if $\sigma_1 \sim_\Gamma \sigma_2$, and $\langle S, \sigma_1 \rangle \Downarrow \sigma_1'$ and $\langle S, \sigma_2 \rangle \Downarrow \sigma_2'$ where $\langle \sigma_1', e_i \rangle \Downarrow v \Leftrightarrow \langle \sigma_2', e_i \rangle \Downarrow v$ for some v for all declassification expressions $e_i \in S$, then $\sigma_1' \sim_\Gamma \sigma_2'$. In short, all pairs of program evaluations that agree on the results of declassified expressions e_i should also agree on other low-visible outputs. Satisfying this condition means that nothing

[3] Similar to the restriction in [11], the theorem holds if checking the invariant $x = \phi$ does not require integral reasoning.

is leaked via low outputs beyond what the declassification expressions already reveal.

We can describe knowledge inference for p in terms of delimited release. Let Γ_p map p-visible variables to L and all remaining variables to H. The set of declassification expressions is the set of output variables (e.g., m in the median example). Now, to see whether local variable y can be inferred by p, we simply label y with L and see whether S still satisfies delimited release. If so, revealing y to p provides no additional information.

The self-composition algorithm described in Section 2.1 is basically checking delimited release. For example, consider the condition presented for the median example:

$$\phi^{sc}_{post} \wedge (\texttt{a1} = \texttt{a1'} \wedge \texttt{a2} = \texttt{a2'} \wedge \texttt{m} = \texttt{m'}) \Rightarrow (\texttt{x1} = \texttt{x1'})$$

The ϕ^{sc}_{post} part captures the semantics of the two executions. The next two equalities are establishing $\sigma_1 \sim_\Gamma \sigma_2$, since they require Alice's two input variables to be equal. The third equality establishes the equality of the declassified output variable m. The final equality x1 = x1' establishes that $\sigma'_1 \sim_\Gamma \sigma'_2$ (where the other low-security variables are known to be equal by virtue of them appearing to the left of the implication, and the program respecting single-assignment semantics).

Alternatives to $\varsigma(S, \phi)$***.*** The role of $\varsigma(S, \phi)$ (Figure 5) is to provide a sound approximation of final states of executing the program S starting from an initial state that satisfies ϕ. We can use other program analyses to get such an approximation. In Section 2.1 we used symbolic execution for this purpose; for our language (Figure 3), which lacks loops, symbolic execution generates equivalent formula as ς.

While $\varsigma(S, \phi)$ as defined in Figure 5 provides a complete approximation of final program states (Theorem 4), for large programs the formula can become prohibitively large. In such cases, we can always trade completeness of the approximation, and use abstract interpretation [8] to provide a sound approximation. With such analyses, our knowledge inference algorithms are still sound, in that if they output y is known to p then $\mathfrak{K}(S, p, y)$, but they lose completeness.

Applying information flow analysis. In the limit, we can use a grossly over-approximating language-based information flow analysis [19] for knowledge inference. Following the formulation relating knowledge inference to delimited release given above, we can label each of party p's input variables as L and all other input variables as H, restricting valid flows in the program as $L \sqsubseteq H$ as usual, while explicitly declassifying the final output when it is returned. Then we can do type inference [18, 23] to determine whether any unlabeled, local variables can safely be given label L, and if so then we know these can be determined solely from knowledge of p's inputs.

Such a type-based analysis is less precise than the semantic analysis we have given to this point. It cannot, for example, infer the knowledge of x1 and x2 in the median example.

```
1  ## variable with suffix A are Alice's inputs,
2  ## with suffix B are Bob's. yd is known to both.
3  int lot_size(int fvA, int cA, int hvA,
4               int fbB, int hbB, int yd)
5    int a, b, c, d, e, f, g, h, i;
6
7    a = 2 * yd;
8    b = a * fvA;
9    c = yd / cA;
10   d = c * hvA;
11
12   e = 2 * yd;
13   f = e * fbB;
14
15   g = f + b;
16   h = hbB + d;
17   i = g / h;
18
19   return sqrt(i); ## integer square root
```

Figure 11. Joint economic lot size example from [14]

As soon as it sees x1 = a1 $\leq$ b1 (Figure 1, line 5) it assumes that there is information flow from both a1 and b1 to x1, and hence, neither Alice nor Bob can determine x1 alone.

However, it is far less expensive than a semantic analysis, and there are some useful examples where such an analysis is enough to establish knowledge facts. Consider the joint economic lot size computation example from Kerschbaum [14], shown in Figure 11. The program computes an order quantity (or lot size) between a buyer (Bob) and vendor (Alice). The buyer's private inputs include the holding cost per item (hbB) and the fixed ordering costs per order (fbB). The vendor's private inputs include the holding cost per item (hvA), the fixed setup costs per order (fvA), and the capacity (cA). Both parties know the yearly demand of the buyer (yd). For vendor Alice, if we label yd, fvA, cA, hvA as L, fbB, hbB as H, and do type inference in an information flow type system, it can infer that a, b, c, d can have label L and are thus known to Alice. Similarly, it can infer that e, f are known to Bob. Using these knowledge facts, the SMC protocol can be optimized to compute lines 7-10 locally on Alice's host, and lines 12-13 locally on Bob's host, leaving only lines 15-17, and 19 to be computed securely.

Adding loops to the programs. SMC programs do not admit loop constructs because in many cases the execution of a loop, specifically the number of times it iterates, can potentially reveal information about parties' input values beyond what is revealed by the output. However, if we can prove that using their own input and output variables, all parties in the secure computation can infer the number of loop iterations, we can allow SMC programs to have loops in them, without compromising security. For example, for a loop `.. i = 0; while(i < n) { ... ++i;} ..` , if `n` is already known to all the parties in the computation, they can infer the number of loop iterations, and hence running this loop in SMC does not compromise security.

Constructive knowledge inference can be useful in this situation. In particular, we can use it to infer loop invariants in terms of known variables for a party, and if we can do so for all the parties, we can admit the loop in SMC.

5. Experiments

In this section, we present an experimental evaluation of our approach. We provide performance measurements for our algorithms on several example programs.

5.1 Implementation

We present evaluation of three implementations of our algorithms – two for the knowledge inference algorithm from Figure 7 that handle linear and non-linear arithmetic respectively, and one for the constructive knowledge inference algorithm from Figures 8 and 9.

Convex polyhedra based implementation. We have implemented the knowledge inference algorithm from Figure 7 using the polyhedra powerset domain as implemented in Parma Polyhedra Library (PPL, v0.11.2) [1]. This approach represents the program postcondition, ϕ_{post}, as a set of convex polyhedra (each of which is a conjunction of linear inequalities), interpreted over real-valued variables. We use polyhedra in the implementation to avoid reasoning about integers as much as possible. To verify the validity of ϕ (line 9 in Figure 7), we check if the negation of ϕ has an integer solution. This corresponds to checking, for every polyhedron/disjunct φ in $\phi_{post} \wedge \phi'_{post} \wedge \phi_k$, that the formulae $\varphi \wedge (y > y')$ and $\varphi \wedge (y < y')$ define convex regions with no real points (quick check) and no integer points (slower check). If so, ϕ is valid. This implementation only handles programs that use linear arithmetic.

Bitvectors based implementation. Our second implementation of the algorithm from Figure 7 uses a bitvector representation of program variables via the Simple Theorem Prover [9] (STP, revision 1671). This implementation handles non-linear arithmetic. It represents formulae (postcondition, ϕ) using logical and arithmetic expressions over fixed-width bit vectors. The validity of ϕ is checked using STP. In addition, STP allows us to construct formulas that relate individual bits of the integer variables, which means we can construct for every $1 \leq i \leq \mathbf{x}$ (for bit width **x**) the formula $\phi_{post} \wedge \phi'_{post} \wedge \phi_k \Rightarrow (y_i = y'_i)$ where y_i designates bit i of variable y. Checking validity of such formulas lets us conclude that parties can potentially infer individual bits, even if they cannot infer whole variables.

Constructive algorithm for boolean variables. We have implemented the constructive knowledge inference algorithm for boolean variables (Figure 8 and Figure 9) using the LLVM compiler infrastructure [16]. We use the Z3 SMT solver [2] for the validity queries.

5.2 Results

We have conducted the experiments on a Mac Pro with two 2.26 GHz quad-core Xeon processors, 16 GB RAM, and running OS X v10.8. The results are in Figure 12.

The top chart shows time taken (in log-scale) by our three implementations, **POLY** (convex polyhedra based), **BVx** (bit-vector based, for **x** as 8, 16, and 32), and **CONS** (constructive algorithm), on several example programs (discussed later). We evaluated **BVx** on all programs, whereas other implementations only on the (linear) median examples. In all programs, we try to infer all variables for both the parties. Additionally, in the case of **BVx**, we also try to infer every intermediate bit.

The bottom chart provides some characteristics of the test cases that contribute to the running times above: the total number of variables in the test cases, and for the linear programs, the number of convex disjuncts in program postcondition (see **POLY** implementation description).

Median example. We consider the joint median computation (Figure 1) for 2, 3, 4, and 5 inputs per party (these versions do not store intermediate integer values `a3`, `b3`, etc. as in Figure 1). Unsurprisingly, the time taken by non-constructive implementations increases with the number of inputs. **POLY** is especially susceptible to the large number of disjuncts in the program postcondition (due to the large number of paths), taking around 24 seconds for analyzing `median5`, up from as little as 0.044 seconds for analyzing `median2`.

For **CONS**, we consider the set Q for Alice as $\{\mathtt{m} \odot a_i\}$ and for Bob as $\{\mathtt{m} \odot b_i\}$, where $\odot \in \{<, \leq, >, \geq, =, \neq\}$, and a_i and b_i range over inputs of Alice and Bob respectively. We used $(c = 2, d = 2)$ for all input sizes. It is able to infer knowledge of all comparisons for all the median programs. However, as the number of candidate predicates (size of Q) increases, the algorithm takes more time. For median with 4 inputs, for example, $|Q_{\text{Alice}}| = |Q_{\text{Bob}}| = 24$, and it takes $\sim$41 seconds to infer all the variables for both parties, as compared to $\sim$2 seconds in the case of 2 inputs per party and $|Q_{\text{Alice}}| = |Q_{\text{Bob}}| = 12$.

We note that at present, our implementation does not aggressively optimize the use of the SMT solver (like caching query responses etc.) that can potentially bring down the inference time since there are lots of redundant validity queries. Moreover, the **CONS** implementation computes ς for every to-be-inferred variable, something that can be optimized as well.

Lot size example. The joint computation of economic lot size in Figure 11 is a non-linear arithmetic example. As described in Section 4, information flow analysis infers that Alice knows `a, b, c, d` and Bob knows `e, f`. Using **BVx**, for **x** as 8, 16, and 32, we infer the same conclusions. In ad-

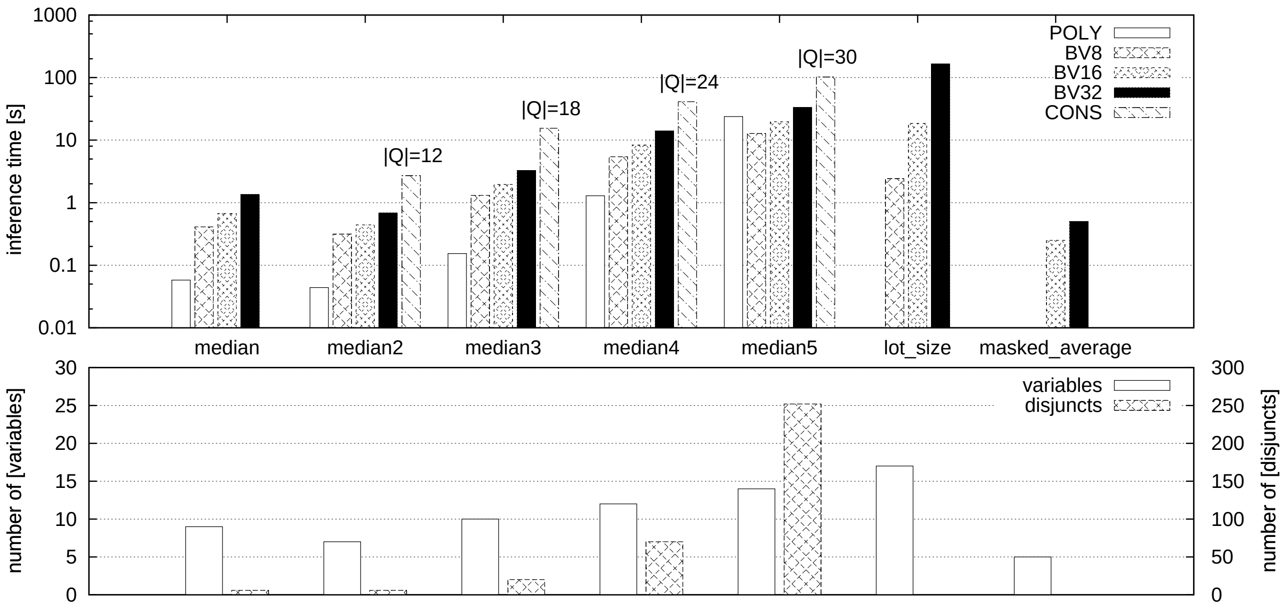

Figure 12. Results (Section 5.2).

dition, various bits of some other variables are inferred. For example, Alice knows bit 1 of `f` and `g`, while Bob knows bit 1 of `b` and `g`. These are due to the multiplications by 2 on lines 9 and 15, resulting in null bits of lowest order.

The performance of **BVx** for this test case naturally decreases as **x** is increased. For **x** as 8, the analysis takes around 2.4 seconds, while for **x** as 32, it takes 165 seconds. Note that a significant portion of this additional time is spent checking a much larger number of bits for partial inference (when complete variables cannot be inferred).

Masked average example. Our final example serves to better demonstrate the inference of bits of variables that cannot be inferred completely. Inference on the scale of bits lets us determine bit-width requirements of a circuit implementing some computation, as well as determine which bits can be revealed ahead of time due to the output of the computation. Consider a `masked_average` function in Figure 13. The function outputs the high-order bits of the average of two 16 (or 32) bit inputs, which are assumed to be 12 bits big. **BVx** implementations for **x** as 16 and 32, analyzes this function in 0.25 and 0.50 seconds respectively.

```
## assume 0 <= a,b < 0x0fff
int masked_average (int a, int b)
  int sum = a + b;
  int avg = sum / 2;
  return (avg & 0xfff0);
```

Figure 13. Masked average

If we only consider the input assumptions (view the program as outputting nothing), then both parties can infer the null values of `sum` at bits 14-16 and of `avg` at bits 13-16 . Additionally, since the function returns all but the lower 4 bits of the average, knowledge inference lets us conclude that bits 6-13 of `sum` and bits 5-12 of `avg` can be inferred given the output. An optimized circuit for this function would (a) reduce the size of sum and avg to 13 and 12 bits respectively, and (b) reveal bits 6-13 of `sum` after computing it, so that the final division circuit can be performed with just 5 bits.

6. Related Work

Knowledge inference in SMC. In contrast to SMC compilers that compute a function as a monolithic secure computation [6, 17], recent research has focused on knowledge inference driven optimized SMC protocols. Huang et. al. [13] identify this limitation of previous compilers, and present a framework for implementing optimized, modular SMC protocols. However, they leave the automatic generation of optimized SMC protocols to future work [12].

Kerschbaum [14] solves the knowledge inference problem using a custom program analysis based on epistemic modal logic inference rules. He shows that his approach works on the median example (Figure 1), and the lot size computation example (Figure 11). Our work can be viewed as a generalization and improvement of his approach, making several advances. First, we formally define the notion of *knowledge* in SMC, and the problem of knowledge inference. Second, we prove our algorithms are sound and (relatively) complete. Moreover, our algorithms are built on top of SMT solvers, thus leveraging recent advances in SMT solving techniques. Indeed, we present experimental measurements to characterize the performance of our algorithms while he does not. To the best of our understanding, the rules presented in [14] fail to handle small perturbations in the programs, e.g. if in the median example, instead of median `m`, the program outputs `m + 1`. Our algorithms, on the other hand, are able to infer knowledge in such cases.

Self-composition and noninterference. Our approach to (non-constructive) knowledge inference takes advantage of the connection between the problem and methods for deciding noninterference-like properties using *self-composition* [4]. As far as we are aware, we are the first to observe that knowledge inference can be reduced to the question of deciding delimited release [20], and we are the first to show how to decide this property using self-composition. Inferring local variables known to p via information flow analysis, as described earlier, is similar to the splitting algorithm employed by Jif/Split [23], which partitions a program to run on multiple hosts. Jif/Split does not employ SMCs, but rather relies on trusted third parties, and employs a simple syntactic algorithm incapable of inferring deeper relationships, e.g., it would not be able to deduce that Alice can infer `x1` and `x2` in the median example.

Template based program verification. Our constructive knowledge inference algorithms (Figure 10, Figure 9, and Figure 8) are inspired by template driven program verification techniques [11, 21]. However, our algorithms take advantage of features specific to our problem. Our templates, instead of having arbitrary structure, have restricted form of $\phi_L \Rightarrow y \wedge y \Rightarrow \phi_R$. For *negative* variables (i.e., variables on the left side of an implication), independent of c and d, we never have to consider more than one lattice, since we always have only one template variable on the left of implication. Second, as mentioned in the inference of ϕ_L earlier, in addition to pruning the subtree of a solution node, we also prune subtrees whose root node has size greater than c. Third, we infer ϕ_L independent of ϕ_R, i.e. solve $\phi_L \Rightarrow y$ separately from $y \Rightarrow \phi_R$, which is different from [21], where *negative* variables are inferred for every permutation of *positive* variables (variables on the right side of an implication). Again, the simple structure of our templates enables us to do so. Finally, we do not attempt to combine solutions in $\mathcal{N}$ to form the strongest solution. We only need a proof of knowledge without worrying about the best such proof.

7. Conclusion

In this paper, we considered the problem of knowledge inference in the context of optimizing secure multi-party computation. We formally defined the notion of knowledge in SMC, and the problems of knowledge inference and constructive knowledge inference. We gave solutions to the knowledge inference problems and proved that our solutions are sound, and characterized conditions under which they are complete. Finally, we implemented our solutions and presented an experimental evaluation of the same.

References

[1] PPL: Parma polyhedral library. `www.cs.unipr.it/ppl`.

[2] Z3 theorem prover. `research.microsoft.com/en-us/um/redmond/projects/z3`.

[3] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the k th-ranked element. In *EUROCRYPT*. Springer, 2004.

[4] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[5] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[6] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, 2008.

[7] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF*, 2008.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, 1976.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[10] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *STOC*, 1987.

[11] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[12] Y. Huang, P. Chapman, and D. Evans. Privacy-preserving applications on smartphones. In *HOTSEC*, 2011.

[13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[14] F. Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] LLVM. `http://llvm.org`.

[17] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: a secure two-party computation system. In *USENIX Security*, 2004.

[18] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.

[19] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 2003.

[20] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *International Symp. on Software Security*, 2004.

[21] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[22] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[23] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *SOSP*, 2001.