Knowledge inference for optimizing and enforcing secure computations

Piotr Mardziel[†], Michael Hicks[†], Jonathan Katz[†], Matthew Hammer[†], Aseem Rastogi[†], and Mudhakar Srivatsa[‡]
University of Maryland, College Park[†] IBM T.J. Watson Research Center[‡]

***Abstract*—We present several techniques that aim to compute the *belief* or *knowledge* a party might have about the values of hidden variables involved in the computation. These techniques can be used for enforcing *knowledge-based security policies* and for optimizing secure multiparty computations.**

I. Introduction

Suppose Bob has some secret data b, and would potentially like to reveal to Alice $y = Q_1(b)$, for some function Q_1, without revealing "too much" about x. More generally, Alice and Bob may each have secrets a and b and would like to learn $z = Q_2(a, b)$, for some function Q_2, without revealing too much about their secrets to each other. There are many useful applications of this basic problem setup, for example:

- Bob might control a sensor network whose features (locations, range, sensing capabilities) are private, but would like to allow Alice to query the network for activity.
- Bob might have a database of sensitive information (e.g., some kind of terrorist watch list), and would like to allow a limited form of query to this database.
- Alice and Bob are moving on a battlefield, and want to determine whether they are within range of one another, but not reveal their exact position.

Our research carried out as part of the ITA has aimed to support scenarios such as these. In particular, we have broadly considered two research questions:

1) How can we write a policy for Alice and Bob that defines when, according to their preferences, executing Q will reveal "too much" about its secret inputs?
2) How can we determine what, or how much, information is revealed about the secret inputs by running of Q? We can use this information to *enforce* the secrecy policy (i.e., if Q reveals too much, don't run it), and to *optimize* the computation in the two (or more) party case.

This paper presents our research into these questions, summarizing previously published work [16], [13], [15], [14].

II. Belief tracking for security enforcement

Our goal is to devise a method whereby each principal can determine whether the output of some function of its secret would reveal too much information about that secret. Suppose that we have two principals, P_1 and P_2, where P_1 has a secret value x_1 and P_2 wishes to compute some function Q of x_1. P_1 will only proceed if P_2 does not learn "too much" about x_1 upon learning the result. The question is: how does P_1 reason what P_2 might learn about x_1 from $Q(x_1)$? To answer this question, we adopted the approach of Clarkson et al. [4]. In their approach, P_2 has a *belief* about the possible values of x_1 and the belief is *revised* upon learning the output of a function over that secret. In our approach, P_1 *estimates* what P_2 might know about x_1 (e.g., that it is uniformly distributed), and then uses Clarkson et al.'s method to determine how much information P_2 might gain from the answer to Q. If this information exceeds a threshold, the query is rejected.

Variables	x	$\in$	**Var**
Integers	n, s, o	$\in$	$\mathbb{Z}$
Rationals	r	$\in$	$\mathbb{Q}$
Arith.ops	aop	$::=$	$+ \mid \times \mid -$
Rel.ops	$relop$	$::=$	$\leq \mid < \mid = \mid \neq \mid \cdots$
Arith.exps	E	$::=$	$x \mid n \mid E_1\ aop\ E_2$
Bool.exps	B	$::=$	$E_1\ relop\ E_2 \mid$
			$B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B$
Statements	Q, S	$::=$	$\mathsf{skip} \mid x := E \mid$
			$\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid$
			$\mathsf{pif}\ r\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid$
			$S_1\ ;\ S_2 \mid \mathsf{while}\ B\ \mathsf{do}\ S$

Fig. 1. Core language syntax

This section reviews Clarkson et al's technique and then presents our application of it to knowledge-based security enforcement, summarizing results presented in more detail elsewhere [15], [14]. Section III generalizes this approach to the case when multiple parties contribute secrets to a joint computation, e.g., computed through a secure multiparty computation (SMC) [18], [8], which is a protocol that simulates the use of a trusted third party to compute the joint computation, but can be carried out directly by the interested parties directly. Section IV shows how inferred knowledge can be used to optimize the underlying SMC.

A. Clarkson et al.'s knowledge estimation

The programming language we use for computations is given in Figure 1. A computation is defined by a statement S whose standard semantics can be viewed as a relation between states: we write $[\![S]\!]\sigma = \sigma'$ to mean that running statement S with input state σ produces output state σ', where states map variables to integers:

$$\sigma, \tau \in \textbf{State} \stackrel{\text{def}}{=} \textbf{Var} \rightarrow \mathbb{Z}$$

Sometimes we consider states with domains restricted to a subset of variables V, in which case we write $\sigma_V \in \textbf{State}_V \stackrel{\text{def}}{=}$

$$
\begin{aligned}
[\![\mathsf{skip}]\!]\delta &= \delta \\
[\![x \ := \ E]\!]\delta &= \delta\,[x \to E] \\
[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta &= [\![S_1]\!](\delta|B) + [\![S_2]\!](\delta|\neg B) \\
[\![\mathsf{pif}\ q\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]\delta &= [\![S_1]\!](q\cdot\delta) + [\![S_2]\!]((1-q)\cdot\delta) \\
[\![S_1\ ;\ S_2]\!]\delta &= [\![S_2]\!]([\![S_1]\!]\delta) \\
[\![\mathsf{while}\ B\ \mathsf{do}\ S]\!] &= \mathrm{lfp}\,[\lambda f:\ \mathbf{Dist}\to\mathbf{Dist}.\ \lambda\delta. \\
&\qquad f\,([\![S]\!](\delta|B)) + (\delta|\neg B)]
\end{aligned}
$$

where

$$
\begin{aligned}
&\delta\,[x \to E] \stackrel{\text{def}}{=} \lambda\sigma.\ \textstyle\sum_{\tau\ |\ \tau[x\to[\![E]\!]\tau]=\sigma}\delta(\tau) \\
&\delta_1 + \delta_2 \stackrel{\text{def}}{=} \lambda\sigma.\ \delta_1(\sigma) + \delta_2(\sigma) \\
&\delta|B \stackrel{\text{def}}{=} \lambda\sigma.\ \mathbf{if}\ [\![B]\!]\sigma\ \mathbf{then}\ \delta(\sigma)\ \mathbf{else}\ 0 \\
&p\cdot\delta \stackrel{\text{def}}{=} \lambda\sigma.\ p\cdot\delta(\sigma) \\
&\|\delta\| \stackrel{\text{def}}{=} \textstyle\sum_\sigma \delta(\sigma) \\
&\mathit{normal}(\delta) \stackrel{\text{def}}{=} \tfrac{1}{\|\delta\|}\cdot\delta \\
&\delta\,\|\,B \stackrel{\text{def}}{=} \mathit{normal}(\delta|B)
\end{aligned}
$$

Fig. 2. Probabilistic semantics for the core language

$V \to \mathbb{Z}$. We will write $\{x_1 = s_1, ..., x_n = s_n\}$ to represent a state σ whose domain is $\{x_1, ..., x_n\}$ such that $\sigma(x_1) = s_1$, $\sigma(x_2) = s_2$, etc. We may also *project* states to a set of variables V:

$$\sigma \upharpoonright V \stackrel{\text{def}}{=} \lambda x \in \mathbf{Var}_V.\ \sigma(x)$$

The language is essentially standard. The semantics of the statement form pif r then S_1 else S_2 is non-deterministic: the result is that of S_1 with probability r, and S_2 with probability $1 - r$.

In our setting, we limit our attention to *queries* in this language. A query is a statement Q that can read, but not write, free variables $x_1, ..., x_n$ (i.e., these are set in the initial state σ), and sets the output to the variable out.

Example 1. As an example, consider the following query:

$$
\begin{aligned}
Q_0 \quad \stackrel{\text{def}}{=} \quad & \mathsf{if}\ x_1 \geq 7 \\
& \quad \mathsf{then}\ out\ :=\ \mathsf{True} \\
& \quad \mathsf{else}\ out\ :=\ \mathsf{False}
\end{aligned}
$$

Given an input state $\sigma = \{x_1 = 3\}$, we have that $[\![Q_0]\!]\sigma = \sigma'$ where $\sigma' = \{x_1 = 3, out = \mathsf{False}\}$.

A belief is represented as a probability distribution, which is conceptually a map from states to positive real numbers representing probabilities (in range $[0, 1]$).

$$\delta \in \mathbf{Dist} \stackrel{\text{def}}{=} \mathbf{State} \to \mathbb{R}+$$

In what follows, we often notate distributions using lambda terms; e.g., we write $\lambda\sigma.\mathbf{if}\ \sigma(x_1) = 3\ \ \mathbf{then}\ 1\ \ \mathbf{else}\ 0$ to represent the point distribution assigning probability 1 to the state σ in which x_1 is 3, and probability 0 to all other states.

Given a principal's initial belief, Clarkson et al. define a mechanism for *revising* that belief according to the output of a query. This works as follows. First, a principal evaluates the query according to its belief using the *probabilistic semantics* given in Figure 2. This semantics is standard (cf. Clarkson et al. [4]) so, due to space constraints, we do not describe it in detail here. It suffices to understand that $[\![S]\!]\delta$ represents probabilistic execution: we write $[\![S]\!]\delta = \delta'$ to say that the distribution over program states after executing S with δ is δ'. We may view δ' as a prediction of the likelihood of the possible final states given some initial distribution of states δ. Upon seeing the actual output of the query, the principal can *revise* this prediction; we write such revision as $[\![S]\!]\delta\,\|\,(out = n)$, where $out = n$ is a boolean expression B and n is the actual observed output. The definition of revision $\delta\,\|\,B$ is given at the bottom of Figure 2. The secret input part of the revised belief can be used as the prior belief for a future query. The revision operation itself is a conditioning, which usually results in a distribution with a mass not equal to 1, followed by a normalization, which produces a real distribution.

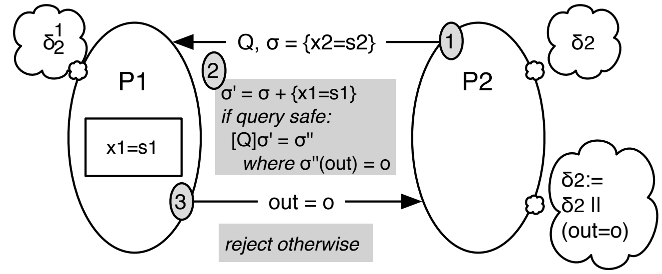

Fig. 3. Asymmetric belief tracking

Returning to Example 1, suppose that x_1 represents P_1's secret value, and P_2's belief δ_2 is as follows

$$\delta_2 \stackrel{\text{def}}{=} \lambda\sigma.\,\mathbf{if}\ \sigma(x_1) < 0\ \mathbf{or}\ \sigma(x_2) \geq 10\ \mathbf{then}\ 0\ \mathbf{else}\ 1/10$$

Thus, δ_2 is a function from states to real numbers implementing a uniform distribution: if x_1's value in σ is between 0 and 9 then σ is given probability $1/10$, otherwise it is given probability 0. To revise δ_2 according to the actual output $out = \mathsf{False}$, principal P_2 first computes $[\![Q_0]\!]\delta_2 = \delta_2'$, which when simplified can be written

$$
\begin{aligned}
\delta_2' \stackrel{\text{def}}{=} \lambda\sigma.&\mathbf{if}\ \sigma(x_1) < 0\ \mathbf{or}\ \sigma(x_2) \geq 10\ \mathbf{then}\ 0 \\
&\mathbf{else\ if}\ \sigma(out) = \mathsf{True}\ \mathbf{and}\ \sigma(x_1) \geq 7\ \mathbf{then}\ 1/10 \\
&\mathbf{else\ if}\ \sigma(out) = \mathsf{False}\ \mathbf{and}\ \sigma(x_1) < 7\ \mathbf{then}\ 1/10 \\
&\mathbf{else}\ 0
\end{aligned}
$$

Revising δ_2' under the assumption that $out = \mathsf{False}$ would produce the following (simplified) distribution:

$$
\begin{aligned}
&\delta_2'\,\|\,(out = \mathsf{False}) \stackrel{\text{def}}{=} \\
&\qquad \lambda\sigma.\,\mathbf{if}\ \sigma(x_1) < 0\ \mathbf{or}\ \sigma(x_2) \geq 7\ \mathbf{then}\ 0\ \mathbf{else}\ 1/7
\end{aligned}
$$

Soundness.: Clarkson et al. show that the probabilistic semantics and revision exactly model the changing belief of an adversary as it learns outputs of the queries, assuming no other channel of information flow exists, and the adversary is rational and has unbounded computational power.

Theorem 2 (Theorem 1 of [4])**.** *A rational, computationally unbounded agent, having belief δ about x_1, updates its belief to δ' after learning output n of a query Q, with no other channels, where δ' is $[\![Q]\!]\delta\,\|\,(out = n)$.*

B. Enforcing knowledge-based security policies

We use Clarkson et al's technique as a key building block for handling the scenario given in Figure 3. Here, in step 1

$\mathsf{tcheck}(q, \delta_i, t_j, x_j) \stackrel{\text{def}}{=}$

1 $\underline{\delta_i} := [\![q]\!]\delta_i$
2 `forall` *possible outputs* o
3 $\quad \hat{\delta}_i := (\underline{\delta_i} \,\|\, (out = o)) \upharpoonright \{x_j\}$
4 $\quad$ `if` $\exists n.\ \hat{\delta}_i(\{x_j = n\}) > t_j$ `then`
5 $\quad\quad$ `return` *reject*
6 `return` *accept*

Fig. 4. Threshold policy decision, tcheck

P_2 sends a query Q and a state σ to P_1. In step 2, P_1 decides whether Q is safe to compute, and if so, executes $[\![Q]\!]\sigma' = \sigma''$, where σ' is σ with the added mapping of x_1 to P_1's secret s_1. In step 3, P_1 sends back the result $o = \sigma''(out)$ if the query was safe, and otherwise rejects the query. P_2 revises its belief δ_2 based on the outcome.

The main question to answer is how P_1 determines whether Q is safe, i.e., whether it "reveals too much information." We propose that principal P_1 assign to its secret a *knowledge threshold* t_1, where $0 < t_1 \leq 1$, interpreted to mean that P_2 should never be certain of P_1's secret with probability greater than t_1. Returning to Example 1, suppose that P_1's knowledge threshold $t_1 = 1/10$ and $x_1 = 3$. Running Q_0 produces False, and P_2's revised belief δ_2' assigns to the state $\{x_1 = 3, out = \mathsf{False}\}$ the probability $1/7$, which exceeds the threshold. As such P_1 ought to reject the query. On the other hand, if the threshold was $1/2$, then the query could be accepted.

Keeping this intuition in mind, here is how the part notated *is the query safe* in Figure 3 is implemented. First, P_1 *estimates* P_2's belief δ_2 about P_1's secret value. We write δ_2^1 to indicate this estimate.[1] Then P_1 calls $\mathsf{tcheck}(Q, \delta_2^1, t_1, x_1)$, the pseudocode for which is given in Figure 4. Here, δ_i is bound to P_1's estimate δ_2^1, while t_j and x_j are bound to t_i and x_i (that is, the *variable name* x_i, not the value it is bound to), respectively.

On line 1, P_1 probabilistically executes $[\![Q]\!]\delta_i$ producing $\underline{\delta_i}$. Then, for each possible output o (line 2), P_1 can revise the belief, $\underline{\delta_i} \,\|\, (out = o)$, from which we can project states to involve only secret x_1, written $\hat{\delta}_i = (\underline{\delta_i}|(out = o)) \upharpoonright \{x_1\}$ (line 3). We explain shortly why every possible output must be considered, rather than just the output for P_1's actual secret value. On line 4, we check whether for o and corresponding revised belief $\hat{\delta}_i$ there exists a possible value n such that $(\hat{\delta}_i)(\{x_1 = n\}) > t_1$. If so, the query Q must be rejected, to avoid leaking too much information (line 5). Otherwise, the query is acceptable (line 6).

If $\mathsf{tcheck}(Q, \delta_2^1, t_1, x_1)$ returns *accept* then P_1 can execute the query, send back the result, and update its estimate δ_2^1 to be $\underline{\delta_2^1} \,\|\, (out = o)$.

Avoiding leakage due to query rejection: Line 2 in Figure 4 requires we consider all possible outputs o. At first glance, doing so seems unnecessarily conservative. For Example 1, suppose that $t_1 = 1/5$ and $\delta_1^2 = \delta_2$; then executing $\mathsf{tcheck}(Q_0, \delta_2^1, t_1, x_1)$ would produce *reject*. But if the actual secret is $x_1 = 3$, then we have already established that answering the query (with False) results in δ_2 being revised to assign $\{x = 3, out = \mathsf{False}\}$ probability $1/7$ which is below the threshold. On the other hand, suppose that x_1 was 8 instead of 3, in which case answering the query with True would cause P_2's revised belief to ascribe probability $1/3$ to $\{x_1 = 8, out = \mathsf{True}\}$, which exceeds the threshold $t_1 = 1/5$. But if P_1 rejects the query, and P_2 knows threshold t_1 it will be able to infer that the only reason for rejection would be that the answer would have been True. Even if t_1 is not known directly, it can be inferred by enough queries to eventually make this sort of determination. P_1 avoids this situation by rejecting any query for which there exists a secret that could be compromised by the answer, even if that does not happen to be its secret. This approach results in P_1 deciding to allow a query or not independetly of his true secret value. Such policy decisions are *simulatable* [9] in that P_2 could have determined on their own whether P_1 will reject the query, hence learning of P_1's decision tells them nothing.

Implementation: We implement the probabilistic semantics given in Figure 2 using *abstract interpretation* [6]. In our technique, we represent a distribution of integers as a *probabilistic polyhedron*, which is roughly a list of linear constraints involving uncertain variables, along with probabilities associated with regions defined by these constraints. The more common implementation approach is to use some form of sampling—basically, just run the program with values sampled from the input distributions, and collect their outputs to produce the output distribution. This approach is more general, but is prohibitively slow. Space constraints preclude a detailed presentation of our implementation, but complete details, along with an extensive performance evaluation, can be found in our prior publications [15].

III. Enforcing knowledge thresholds for SMC

In this section we show how to generalize knowledge-based enforcement to the multi-secret setting. In this setting, there are N principals, $P_1, ..., P_N$ each with a secret $x_1 = s_1, ..., x_N = s_N$. Each P_i maintains a belief δ_i about the possible values of the other participating principals' secrets. In addition, each P_i has a knowledge threshold t_i that bounds the certainty that the other principals can have about its secret's value. Our goal will be to define how the principals should decide whether to safely participate in the joint computation Q, executed as an SMC. We discuss two possible methods for doing so: the *belief set* method (Section III-B) and the *SMC belief tracking* method (Section III-C).

A. Running example

Suppose we have three principals, P_1, P_2, and P_3, each with a net worth $x_1 = 20$, $x_2 = 15$, and $x_3 = 17$, in millions of dollars, respectively. Suppose they wish to compute Q_1 which

[1] How P_1 comes by this estimate is beyond the scope of this paper, but we point out that for many kinds of data, good estimates are easy to come by. For example, generic distributions over personal information like gender, birthday, social security number, income, etc. can be gained from census data or other public and private repositories (e.g., Facebook demographics).

determines whether P_1 is the richest:

$$Q_1 \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{if}\ x_1 \geq x_2 \wedge x_1 \geq x_3 \\ \quad \mathsf{then}\ out := \mathsf{True} \\ \quad \mathsf{else}\ out := \mathsf{False} \end{array}$$

Using the idealized view, each of P_1, P_2, and P_3 can be seen as sending their secrets to P_T, which initializes σ such that $\sigma(x_1) = 20$, $\sigma(x_2) = 15$, and $\sigma(x_3) = 17$. Running Q_1 using σ produces an output state σ' such that $\sigma'(out) = \mathsf{True}$.

Now suppose that P_1 believes that both P_2 and P_3 have at least \$10 million, but less than \$100 million, with each case equally likely. Thus principal P_1's belief is defined as

$$\begin{array}{rl} \delta_1 \stackrel{\text{def}}{=} \lambda\sigma.\textbf{if} & \sigma(x_2) < 10 \textbf{ or } \sigma(x_2) > 100 \textbf{ or} \\ & \sigma(x_3) < 10 \textbf{ or } \sigma(x_3) > 100 \textbf{ or} \\ & \sigma(x_1) \neq 20 \textbf{ then } 0 \textbf{ else } 1/8281 \end{array}$$

States which ascribe either x_2 or x_3 a net worth outside the expected range, or ascribe x_1 to the wrong value, are considered impossible, and every one of the remaining 8281 (that is 91×91) states is given probability $1/8281$. The beliefs of P_2 and P_3 are defined similarly.

Belief revision proceeds as before: once P_T performs the computation and sends the result, each P_i revises its belief. For our example query Q_1, principal P_1 would perform $[\![Q_1]\!]\delta_1 = \delta_1''$ and since the output of the query is True, then revision produces $\delta_1' = [\![Q_1]\!]\delta_1 | (out = \mathsf{True})$. This revised belief additionally disregards states that ascribe x_2 or x_3 to values greater than P_1's own wealth, which is \$20M:

$$\begin{array}{rl} \delta_1' \stackrel{\text{def}}{=} \lambda\sigma.\textbf{if} & \sigma(x_2) < 10 \textbf{ or } \sigma(x_2) > 20 \textbf{ or} \\ & \sigma(x_3) < 10 \textbf{ or } \sigma(x_3) > 20 \textbf{ or} \\ & \sigma(x_1) \neq 20 \textbf{ then } 0 \textbf{ else } 1/121 \end{array}$$

The revised beliefs of P_2 and P_3 will be less specific, since each will simply know that P_1's wealth is at least their own and no less than the rest of the parties.

B. Knowledge-based security with belief sets

Now we wish to generalize threshold enforcement, as described in Section II-B, to SMC. In the simpler setting P_1 maintained an estimate δ_2^1 of P_2's belief δ_2. In the SMC setting we might imagine that each P_j maintains a belief estimate δ_i^j and then performs $\mathsf{tcheck}(q, \delta_i^j, t_j, x_j)$ for all $i \neq j$. If each of these checks succeeds, then P_j is willing to participate.

The snag is that P_j cannot accurately initialize δ_i^j for all $i \neq j$ because it cannot directly represent what P_i knows about x_i—that is, its exact value. So the question is: how can P_j estimate the potential gain in P_i's knowledge about x_j after running query without knowing x_i?

One approach to solving this problem, which we call the *belief set* method, is the following. P_j follows roughly the same procedure as above, but instead of maintaining a single distribution δ_i^j for each remote party P_i, it maintains a *set* of distributions where each distribution in the set applies to a particular valuation of x_i. As a first cut, suppose that P_j initializes this set to be as follows:

$$\Delta_i^j \stackrel{\text{def}}{=} \{\delta_i^j \,\|\, (x_i = v) \;:\; v = \sigma(x_i), \sigma \in \mathit{support}(\delta_j \upharpoonright \{x_i\})\}$$

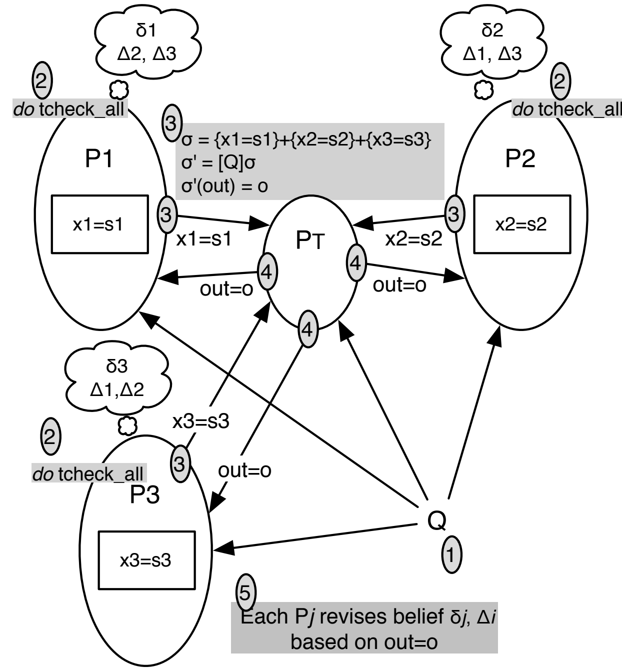

Fig. 5. Threshold enforcement for SMC using belief sets

$\mathsf{tcheck\_all}(q, j) \stackrel{\text{def}}{=}$

`forall` $i \in 1..n$ `with` $i \neq j$
  $\mathsf{tcheck}(q, \Delta_i, t_j, x_j)$
`if` *all threshold checks succeed* `then`
  agree to participate
`else`
  refuse to participate

Fig. 6. $\mathsf{tcheck\_all}$ check for belief set enforcement

Thus Δ_i^j is a set of possible distributions, one per possible valuation of x_i that P_j thinks is possible according to its belief δ_j.

However, this method for initializing the set is not quite expressive enough, since it may fail to take into account correlations among beliefs of multiple principals. For example, if it were known (by all principals) that only one of the principals in the running example can have secret value equal to 15, then P_2 would know initially, based on this own secret $x_2 = 15$, that P_1's value x_1 cannot be 15. However, P_1 cannot arrive at this conclusion without knowing x_2, which is, of course, outside of its knowledge initially.

Therefore, we define the initial belief set using a distribution δ over *all* principals' secret data which sufficiently captures any correlations in those secrets. Such a distribution can then be used, given some valuations of secret variables, to derive what a principal's initial belief would be.

$$\Delta_i \stackrel{\text{def}}{=} \{\delta \,\|\, (x_i = v) \;:\; v = \sigma(x_i), \sigma \in \mathit{support}(\delta \upharpoonright \{x_i\})\}$$

Since we are starting from a globally held belief δ, there is no need to distinguish Δ_i^j from Δ_i^k—they are the same Δ_i.

Now each P_j follows the procedure depicted in Figure 5 for the idealized view (with a trusted principal P_T). First, Semantics

$$[\![S]\!]\Delta = \{[\![S]\!]\delta \ : \ \delta \in \Delta\}$$

Operations

$$\begin{array}{ll}
\Delta \upharpoonright V & \stackrel{\text{def}}{=} \{(\delta \upharpoonright V) \ : \ \delta \in \Delta\} \\
\mathit{normal}(\Delta) & \stackrel{\text{def}}{=} \{\mathit{normal}(\delta) \ : \ \delta \in \Delta \ , \ \|\delta\| > 0\} \\
\Delta \,\|\, B & \stackrel{\text{def}}{=} \mathit{normal}(\{(\delta | B) \ : \ \delta \in \Delta\}) \\
\Delta(\sigma) & \stackrel{\text{def}}{=} \max_{\delta \in \Delta} \delta(\sigma)
\end{array}$$

Fig. 7. Probabilistic semantics using sets of distributions

the principals agree on the query Q. Second, each principal P_j performs the threshold check $\mathsf{tcheck\_all}(Q, j)$, whose code is given in Figure 6. Notice that calls to $\mathsf{tcheck}(...)$ on line 2 are with the set Δ_i, rather than a single distribution δ_i^j. The definitions of the operations in the pseudocode in Figure 4, when applied to sets Δ rather single elements δ, are defined in Figure 7. In all but the last case, these operations are just straightforward liftings of the operations on single distributions. For $\Delta(\sigma)$, we return the highest probability for σ of those ascribed to it by distributions in Δ, to assure that our decision to participate or not is safe. Also note that we will always be dealing with non-empty Δ, hence the maximum probability is sufficiently defined. On the other hand, the normalization procedure for distributions δ is only well defined whenever $\|\delta\| > 0$. Because of this, we make sure the normalization for distribution sets only normalizes the normalizable distributions, and discards the rest. The way in which some member distributions of Δ could become non-normalizable, that is, having mass of 0, is by way of the conditioning operation, where the condition is inconsistent with all possible states in the distribution.

In the third step, if the query is acceptable for all P_j, each sends its secret $x_j = s_j$ to P_T, which executes Q using the secret state σ constructed from each secret. Fourth, the result o is sent back to each principal. Finally, as usual, P_1 revises each of its estimates Δ_i and its own belief δ_j. Note that all principals make the same update for Δ_i, hence there really is only one Δ_i, known by all, estimating P_i's knowledge.

While we have depicted this procedure in the idealized view of SMC, it is easy to see that we can simply implement steps 3 and 4 as a normal SMC and the remainder of the procedure is unchanged.

Soundness: The belief set procedure is *sound*, in that for all P_i, participating or not participating in a query will never increase another P_j's certainty about P_i's secret above its threshold t_i. The proof can be found elsewhere [13].

C. SMC belief tracking for knowledge-based security

Now we present an alternative to the belief set method, in which the decision to participate or not, involving checking thresholds after belief revision, takes place *within the SMC itself*. As such, we call this method *SMC belief tracking*. Once again we present the algorithm using the ideal world with a trusted third party P_T. The steps are shown in Figure 8. The first step is that each P_i presents its secret $x_i = s_i$ to P_T, along with the collective belief δ. Principal P_T then initializes the computation state by calling $\mathsf{init\_SMC}(s_1...s_N, \delta)$, given in Figure 9. On line 1, this code initializes the secret state σ_s that contains all of the secrets. On lines $2..(n{+}1)$, it initializes each principal P_i's belief as in the belief set case, by specializing δ with the knowledge unique to P_i. It also initializes each threshold t_i to r_i.

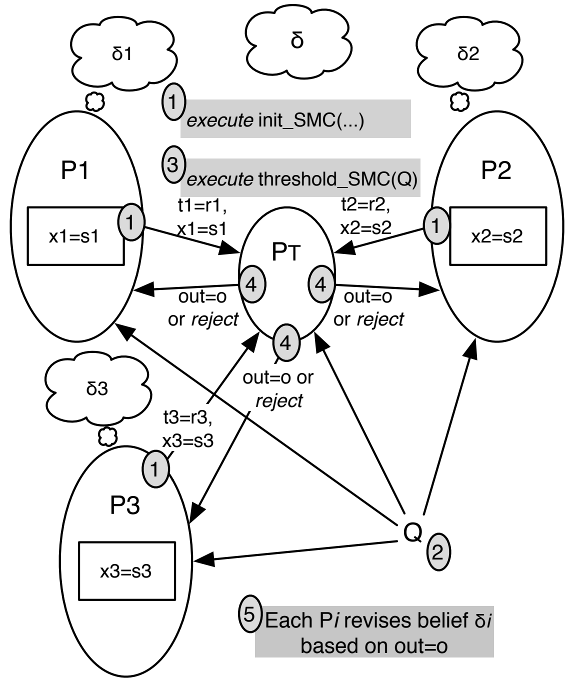

Fig. 8. SMC belief tracking scenario (ideal view)

$\mathsf{init\_SMC}(s_1...s_N, r_1...r_N, \delta) \stackrel{\text{def}}{=}$

1 $\sigma_s := \{x_1 = s_1, ..., x_N = s_N\}$
2 $\delta_1 := \delta \,\|\, (x_1 = s_1);\ t_1 := r_1$
...
$n{+}1$ $\delta_N := \delta \,\|\, (x_N = s_N);\ t_N := r_N$

$\mathsf{threshold\_SMC}(Q) \stackrel{\text{def}}{=}$

1 $o := ([\![Q]\!]\sigma_s)(\mathit{out})$
2 `forall` $j \in 1..n$
3 `forall` $i \in 1..n$ `with` $i \neq j$
4 $\mathsf{tcheck}(Q, \delta_j, t_i, x_i)$
5 `if` *all threshold checks succeed* `then`
6 $\delta_j := [\![Q]\!]\delta_j \,\|\, (\mathit{out} = o)$
7 `return` o `to` P_j
8 `else`
9 `return` *reject* `to` P_j

Fig. 9. SMC belief tracking (ideal view)

In step 2 (of the diagram), the query Q is made available to P_T, which then runs (in step 3) $\mathsf{threshold\_SMC}(Q)$, also shown in Figure 9. On line 1 we compute the actual output o for the query, based on the secret state. On line 2 we loop over each principal P_j. The remainder of the code aims to decide whether answering the query and sending the result to P_j would reveal too much information; if not, we send P_j the answer o (line 7) and otherwise we reject.

Returning to the body of the loop, the next step is to make sure that for every P_i (line 3) its threshold check (Figure 4) will not reject P_j. That is, given the query q and the estimated knowledge of P_j, we make sure that the answer to the query will not reveal too much about P_i's secret x_i (where by "too much" we mean P_j's certainty about P_i's possible secret exceeds threshold t_i). Assuming all P_i threshold checks succeed (line 5), we then revise the P_j's belief according to the output o (line 6), which we then send to P_j (step 4 in the diagram). No revision is done on P_j's belief if the query is rejected for P_j. Finally, each principal revises its own belief δ_j based on the output.

We can repeat steps 2–5 for each subsequent query Q', and P_T will use any beliefs δ_j revised from the run of Q. By performing threshold_SMC as part of an SMC, no participant P_i is ever shown the opposite's secret, and yet an accurate determination is made for each about whether to participate.

Importantly, the fact that P_j receives a proper answer or *reject* is not (directly) observed by any other P_j; such an observation could reveal information to P_j about x_i. For example, suppose $Q_2 \stackrel{\text{def}}{=} x_1 \leq x_2$ and both secrets are (believed to be) between 0 and 9. If $x_2 = 0$ then $[\![Q_2]\!]\sigma_s$ will return True only when x_1 is also 0. Supposing $t_1 = 3/5$, then P_2 should receive *reject* since there exists a valuation of x_1 (that is, 0) such that P_2 could guess x_1 with probability greater than $3/5$. Similar reasoning would argue for reject if $x_2 = 9$, but acceptance in all other cases. As such, if P_1 observes that P_2 receives *reject*, it knows that x_2 must be either 0 or 9, independent of t_2; as such, if $t_2 < 1/2$ we have violated the threshold by revealing the result of the query.

This asymmetry means that threshold_SMC may return a result for one participant but not the other, e.g., P_1 might receive *reject* because t_2 is too low while P_2 receives the actual answer because t_1 is sufficiently high. Nonetheless each P_i's threshold will be respected.

Lacking a trusted third party in the real world, the participants can use secure multi-party computation and some standard cryptographic techniques to implement P_T's functionality amongst themselves. The interesting part is that we need a way for the participants to maintain P_T's state amongst themselves while preserving its secrecy. This can be done using *secret shares* of Σ_T distributed among the parties. The query-evaluation procedure would receive these shares along with the normal inputs, and then distribute the revised versions along with the normal outputs.

Soundness: Once again we can prove a soundness property of this approach, assuring that our approach approximates learning by the participants as it would actually happen, and that query rejection reveals nothing about other parties' inputs [13].

D. Implementation notes

We have developed a proof-of-concept implementation of both methods, and with a series of experiments we have found that SMC belief tracking is strictly more precise (in that fewer queries will be rejected) than belief sets. On the other hand, SMC is known to be very slow, and so implementing KBSE as an SMC could be quite costly. Further details can be found in our prior paper [13]. We leave exploration of implementation strategies to future work.

```
## 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;
```

Fig. 10. Joint median computation example [10]. `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.

IV. Optimizing multi-party computations

So far we have analyzed the function Q with a security mindset: there is a danger in running Q if an observing party would infer that a particular input to Q is sufficiently likely. Now we consider such inferences from the point of view of optimizing performance in the setting of SMC. In particular, we observe that when the SMC reveals the final output, one or all parties may be able to infer the results of intermediate computations, given knowledge of their own inputs and the function being computed, *no matter the inputs of the additional participating parties*. When such inference is possible, the inferable intermediate results need not be cryptographically concealed. As it turns out, turning a single monolithic SMC into several smaller SMCs (by revealing intermediate results early) can have a dramatic impact on performance; Kerschbaum [10] has measured improvements of up to $30\times$ for the median example in Figure 10, discussed shortly. Revealing inferable results does not change the *knowledge profile* of the protocol: If the party will eventually know the intermediate result when given the final output then revealing it earlier does not change what is known to whom.

In this section, we present the main idea of a technique we call *knowledge inference* that aims to find which intermediate variables can be revealed early. We have also developed a technique we call *constructive knowledge inference* that not only identifies which intermediate variables can be inferred, but also exactly the function of the inputs and outputs that defines them. We refer the reader to our prior paper for details of both techniques [16].

A. Knowledge inference

In our setting, party P knows the (deterministic) program Q, his own input (set) s, and his output o.[2] We say party P can *infer* the value of local variable y in Q if there exists a function F such that $y = F(s, o)$ in all runs of Q. Another way of putting it is that no matter the values of P's inputs or

[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.

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

Fig. 11. Path conditions for secure median

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 Q 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. We have developed approaches to both tasks, but here only describe the first.

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 Q that agree on the valuations of variables in s 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 s and o, and thus a function F exists such that $F(s, o) = y$. 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 Q (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.

One program analysis we might use to produce ϕ_{post} is symbolic execution [11]. Each feasible program path is characterized by a *path condition* φ_i, which is a set of predicates relating the program variables. The path conditions 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 10, there are four possible paths, having the path conditions given in Figure 11.

Consider the first path condition φ_1. Conceptually, it describes the program path in which `then` branch of both conditionals (lines 6 and 9) is taken. The remaining three paths constitute the other three possible branching combinations. 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 $\phi_{pre} \stackrel{\text{def}}{=} \mathtt{a1} < \mathtt{a2} \wedge \mathtt{b1} < \mathtt{b2} \wedge \mathtt{a1} \neq \mathtt{b1} \wedge \mathtt{a1} \neq \mathtt{b2} \wedge \mathtt{a2} \neq \mathtt{b1} \wedge \mathtt{a2} \neq \mathtt{b2}$.

The next step is to prove that any two runs of the program Q that agree on variables known to P will also agree on the value of y. This statement is a *2-safety property* [5], and we can prove it using a technique called self-composition [3].

```
## 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');
```

Fig. 12. Median computation composed with itself.

The idea is to reduce this two-run condition on program Q to a condition on a single run of a *self-composed program* Q_c, which is the sequential composition of Q with itself, with the second copy of Q'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 12 shows self composition of the median function of Figure 10. We write $\mathtt{median}'$ for the function `median` but with the local variables renamed to $\mathtt{x1}', \mathtt{x2}', \ldots$. 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 $\mathtt{a1}, \mathtt{a2}, \mathtt{m}$ 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.

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

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

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

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 $\phi^{sc}_{post} = \varphi^{sc}_1 \vee \ldots \vee \varphi^{sc}_{16}$ as the disjunctive normal form of the conjunction of the post condition ϕ_{post} with its primed version

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`.

As it turns out, the knowledge inference question bears a close resemblance to deciding the property of *delimited release* [17].

B. Implementation and experiments

We have implemented the knowledge inference algorithm 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, 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. We also have an implementation based on bitvectors that can infer whether particular bits of variables are known, rather than entire variables.

We have used our implementation on the median example given earlier and several other examples. We find that inference times are on the order of seconds, or tens of seconds, for small programs. More details can be found in our prior paper [16].

V. Related and Future Work

a) Belief tracking (asymmetric): Others have considered how an adversary's knowledge of private data might be informed by a programs output. Ours differs in having a stronger security criterion considering the worst case outcome, rather than an expectation. The idea of strengthening of an entropy measure by eliminating the expectation has been briey considered by Köpf and Basin [12]. The other distinguishing feature of our approach is that we keep an on-line model of adversary knowledge according to prior, actual query results. The core of our methodology relies on probabilistic computation. A variety of tools exist for specifying random processes as computer programs and performing inference on them. Our approach is different than prior work in its emphasis on *soundness*: any approximations made will only reject safe queries, and never accept unsafe ones.

b) Belief tracking (for SMC): Almost all prior work on SMC treats the function Q being computed by the parties as given, and is unconcerned with the question of whether the parties should agree to compute Q in the first place. Dwork et al. [7] show that if f is a differentially private function, then the process of running an SMC protocol that computes f is also differentially private (at least in a computational sense). The security goal we are aiming for is incomparable with that of differential privacy.

c) Knowledge inference: Kerschbaum [10] 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 10), and a *lot size computation* (which we also experiment with). Our work improves on his in several ways. 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.

d) Next steps: We are actively working to extend these threads of work. For example, we are working to extend belief tracking to support real-valued secrets and continuous distributions, using Mathematica as a back end. In addition, we are extending our theoretical development to account for knowledge gained about secrets whose values might change between queries, and inferences about future values based on past changes. Finally, we are working on a compiler for SMCs that incorporates knowledge inference.

Acknowledgments.: This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

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] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, 2004.
[4] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *J. Comput. Secur.*, 17(5), 2009.
[5] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, 2008.
[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN Symposium on the Principles of Programming Languages (POPL)*, 1977.
[7] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Advances in Cryptology — Eurocrypt 2006*, volume 4004 of *LNCS*, pages 486–503. Springer, 2006.
[8] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.
[9] Krishnaram Kenthapadi, Nina Mishra, and Kobbi Nissim. Simulatable auditing. In *PODS*, 2005.
[10] Florian Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.
[11] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
[12] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.
[13] Piotr Mardziel, Michael Hicks, Jonathan Katz, and Mudhakar Srivatsa. Knowledge-oriented secure multiparty computation. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2012.
[14] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, pages 114–128, June 2011.
[15] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, January 2013. To appear.
[16] Aseem Rastogi, Piotr Mardziel, Matthew Hammer, and Michael Hicks. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2013.
[17] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *International Symp. on Software Security*, 2004.
[18] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.