Automating Efficient RAM-Model Secure Computation

Chang Liu Yan Huang Elaine Shi Jonathan Katz Michael Hicks
University of Maryland
College Park, Maryland 20742
Email: {liuchang, yhuang, elaine, jkatz, mwh}@cs.umd.edu

***Abstract*—RAM-model secure computation addresses the inherent limitations of circuit-model secure computation considered in almost all previous work. Here, we describe the first *automated* approach for RAM-model secure computation in the semi-honest model. We define an intermediate representation called SCVM and a corresponding type system suited for RAM-model secure computation. Leveraging compile-time optimizations, our approach achieves order-of-magnitude speedups in comparison with circuit-model secure computation for large datasets, as well as naive implementations of RAM-model secure computation.**

I. Introduction

Secure computation allows mutually distrusting parties to make collaborative use of their local data without harming privacy of their individual inputs. Since Yao's seminal paper [29], research on secure two-party computation—especially in the semi-honest model we consider here—has flourished, resulting in ever more efficient protocols [6], [10], [15], [30] as well as several practical implementations [7], [11]–[13], [16], [20]. Since the first system for general-purpose secure two-party computation was built in 2004 [20], efficiency has improved from being able to compute roughly 6000 boolean gates per second to 10^8 gates per second [6], [13].

Almost all previous implementations of general-purpose secure computation assume the underlying computation is represented as a *circuit*. The efficiency of circuit-model secure computation is limited by several factors. The first challenge is dealing with *dynamic memory accesses* to an array in which the memory location being read/written depends on secret inputs. In the circuit model of computation, such accesses must be handled by considering a circuit that takes the entire array as input, resulting in a huge circuit and prohibitive cost. More generally, although theorists are used to working with circuits, the Random Access Machine (RAM) model is what programmers are used to, and more naturally reflects the von Neumann architecture of today's computing platforms. Generic approahces for translating RAM programs into circuits incur, in general, $\tilde{O}(T^3)$ blowup in efficiency, where T is an upper bound on the program's running time.

To address the above limitations, researchers have more recently considered secure computation that works directly in the RAM model [10], [19]. The key insight is to rely on Oblivious RAM (ORAM) [8] to enable dynamic memory access with (poly-)logarithmic cost, while preventing information leakage through memory-access patterns. Gordon et al. [10] observed a significant advantage of RAM-model secure computation (RAM-SC) in the setting of *repeated sublinear-time queries* (e.g., binary search) on a large database. By amortizing the setup cost over many queries, RAM-SC can achieve *amortized* cost asymptotically close to the run-time of the underlying program in the insecure setting.

A. Our Contributions

We continue work on secure computation in the RAM model, with the goal being to provide a complete system that takes a program written in a high-level language and compiles it to a protocol for secure two-party computation of that program. (Note that Gordon et al. [10] do not provide such a compiler; they only implement RAM-model secure computation for the particular case of binary search.) Toward this end, we

- Define an *intermediate representation* (which we call SCVM) suitable for efficient two-party RAM-model secure computation;
- Develop a *type system* ensuring that any well-typed program will generate a RAM-SC protocol secure in the semi-honest model, if all subroutines are implemented with a protocol secure in the semi-honest model.
- Build an *automated compiler* that transforms programs written in a high-level language into a secure two-party computation protocol, and integrate compile-time optimizations crucial for improving performance.

We use our compiler to compile several programs including Dijkstra's shortest-path algorithm, KMP string matching, binary search, etc. For moderate data sizes (up to the order of a million elements), our evaluation shows a speedup of *1–2 orders of magnitude* as compared to standard circuit-based approaches for securely computing these programs. We expect the speedup to be even greater for larger data sizes.

B. Techniques

As explained in Sections II-A and III, a naive implementation of RAM-SC would entail placing all data and instructions inside a single Oblivious RAM. The secure evaluation of one instruction would then incur *i)* fetching instruction and data from ORAM; and *ii)* securely executing the instruction using a universal next-instruction circuit (similar to a machine's ALU

unit). This approach is costly since each step must be done using a secure-computation sub-protocol.

An efficient representation for RAM-SC. Our type system and SCVM intermediate representation are designed to be capable of expressing RAM-SC tasks efficiently, such that usage of expensive next-instruction circuits can be avoided, and ORAM operations can be minimized without risking security. These language-level capabilities allow our compiler to apply compile-time optimizations that would otherwise not be possible. Thus, we not only obtain better efficiency than circuit-based approaches, but we also achieve *order-of-magnitude* performance improvements in comparison with straightforward implementations of RAM-SC (see Section VI-C).

Program-trace simulatability. A well-typed program in our language is guaranteed to be both *instruction-trace oblivious* and *memory-trace oblivious*. Instruction-trace obliviousness ensures that the values of the program counter during execution of the protocol do not leak information about secret inputs other than what is revealed by the output of the program. As such, the parties can avoid securely evaluating a universal next-instruction circuit, but can instead simply evaluate a circuit corresponding to the current instruction. Memory-trace obliviousness ensures that memory accesses observed by one party during the protocol's execution similarly do not leak information about secret inputs other than what is revealed by the output. In particular, if access to some array does not depend on secret information (e.g., a linear scan of the array is performed), then the array need not be placed into ORAM.

We formally define the security ensured by our type system as *program-trace simulatability*. We define a mechanism for compiling programs to protocols that rely on certain ideal functionalities. We prove that if every such ideal functionality is instantiated with a semi-honest secure protocol computing that functionality, then any well-typed program compiles to a semi-honest secure protocol computing that program.

Additional language features. Our language also supports several other useful features. First, it permits *reactive* computations by allowing output not only at the end of the program's execution, but also while it is in progress. Our notation of program-trace simulatability also fits this reactive model of computation.

SCVM also integrates state-of-the-art optimization techniques that have been suggested previously in the literature. For example, we support public, local, and secure modes of computation, a technique recently proposed (in the circuit model) by Kerschbaum [15] and formalized by Rastogi et al. [24]. In this way, our compiler can identify and encode portions of computation that can be safely performed in the clear or locally by one of the parties, without incurring the cost of a secure-computation sub-protocol.

Our SCVM intermediate representation generalizes circuit-model approaches. For programs that do not rely on ORAM, our compiler effectively generates an efficient circuit-model secure-computation protocol. This paper focuses on the design of the intermediate representation language and type system for secure RAM-model computation, as well as the compile-time optimization techniques we apply. Our work is complementary to several independent, ongoing efforts focused on improving the cryptographic back end.

II. Background and Related Work

A. RAM-Model Secure Computation

In this section, we review some background for RAM-model secure computation. Our treatment is adapted from that of Gordon et al. [10], with notations adjusted for our purposes.

A key underlying building block of RAM-model secure computation is *Oblivious RAM (ORAM)*, implementations of which were initially proposed by Goldreich and Ostrovsky [8] and later improved in a sequence of works [9], [17], [26]–[28]. ORAM is a cryptographic primitive that hides memory-access patterns by randomly reshuffling data in memory. With ORAM, each memory read or write operation incurs $poly \log n$ actual memory accesses. In the standard setting where a server holds the ORAM array, all entries in the array are encrypted with a key stored by the client. (This was also done in [10] in order to ensure that one party holds only $O(1)$ state.) In this work, instead, we secret-share all data stored in the ORAM array using a simple XOR-based secret-sharing scheme. A memory access thus requires each party to access the corresponding element of their array, and the value stored at that position is then obtained by XORing the values read by each of the parries.

We introduce some notation to describe the execution of a RAM program. We let `mem` refer to the memory maintained by the program. We let $(\texttt{pc}, \texttt{raddr}, \texttt{waddr}, \texttt{wdata}, \texttt{reg}) \leftarrow U(\texttt{I}, \texttt{reg}, \texttt{rdata})$ denote a single application of the *next-instruction circuit* (i.e., the CPU's ALU), taking as input the current instruction `I`, the current register contents `reg`, and a value `rdata` (representing a value just fetched from memory), and outputting the next value of the program counter `pc`, an updated register file `reg`, a read address `raddr`, a write address `waddr`, and a value `wdata` to write to location $\texttt{mem}[\texttt{waddr}]$.

$\texttt{mem}[i]$	the memory value at index i
`pc`	the current program counter
`reg`	an $O(1)$-sized set of registers
`I`	an instruction
U	the next-instruction circuit
`rdata`	the last value read from memory
`wdata`	the value to write to memory
`raddr`	a read address
`waddr`	a write address

In RAM-model secure 2-party computation, the entire memory (containing both program instructions and data) is placed in ORAM, and the ORAM is secret shared between the two participating parties as discussed above. All CPU states, including `pc`, `reg`, `I`, `rdata`, `wdata`, are secret-shared between the two parties. The memory addresses `raddr` and `waddr` are revealed to the parties during the course of accessing memory; using ORAM ensures that revealing these addresses does not reveal information about sensitive inputs.

A generic RAM-model secure computation protocol proceeds as in Figure 1. In general, each step of the computation must be done using some secure-computation subprotocol, and indeed this is what is done in the work of Gordon et al. In

/ Variables in blue background (e.g., `var`) are secret-shared between the two parties. Memory addresses `raddr` and `waddr` are revealed in the clear to both parties. */*

For $i = 1, 2, \ldots, t$ where t is the maximum run-time of the program:

instr. fetch phase:	`I` ← `ORAM.Read(mem[pc])`	*//Protocol SC-ORAM*
CPU phase:	(`pc`, `raddr`, `waddr`, `wdata`, `reg`) ← U(`I`, `reg`, `rdata`)	*//Protocol SC-U*
data read phase:	`rdata` ← `ORAM.Read(mem[raddr])`	*//Protocol SC-ORAM*
data write phase:	`ORAM.Write(mem[waddr], wdata)`	*//Protocol SC-ORAM*

Fig. 1: Generic RAM-model secure computation. The parties repeatedly perform secure computation to obtain the next instruction I, execute that instruction, and then read/write from/to main memory. In general, a secure-computation subprotocol must be used to carry out each step, but in our case this is only needed for the CPU phase.

	Scenario	Potential benefits of RAM-model secure computation
1	Repeated sublinear queries over a large dataset (e.g., binary search, range query, shortest path query)	• Amortize preprocessing cost over multiple queries • Achieve *sublinear* amortized cost per query
2	One-time computation over a large dataset	Avoid paying $O(n)$ cost per dynamic memory access

TABLE I: Two main scenarios and advantages of RAM-model secure computation

our case, because we using a simple XOR-based secret-sharing scheme to share the ORAM memory between the parties, all steps except the CPU phase can be carried out by the parties locally, without the need for secure computation.

Scenarios for RAM-model secure computation. While Gordon et al. describe RAM-model secure computation mainly is the amortized setting, where repeated computations are carried out starting from a single initial dataset, we note that RAM-model secure computation can also be meaningful for one-time computation on large datasets, since a straightforward RAM-to-circuit compiler would incur linear (in the size of dataset) overhead for every dynamic memory access whose address depends on sensitive inputs. Table I summarizes the two main scenarios for RAM-model secure computation, and potential advantages of using the RAM model in these cases.

B. Other Related Work

Automating and optimizing circuit-model secure computation. As mentioned earlier, a number of recent efforts have focused on automating and optimizing secure computation in the circuit model. Intermediate representations for secure computation have been developed in the circuit model, e.g., [16]. Mardziel et al. [21] proposed a way to reason about the amount of information declassified by the result of a secure computation, and Rastogi et al. [24] used a similar analysis to infer intermediate values that can be safely declassified without revealing further information beyond what is also revealed by the output. These analyses can be applied to our setting as well (e.g., for optimization or policy enforcement).

Zahur and Evans [31] also attempted to address some of the drawbacks of circuit-model secure computation. Their approach, however, focuses on designing efficient circuit structures for specific data structures, such as stacks and queues, and do not generalize for arbitrary programs. Many of the programs we use in our experiments are not supported by their approach.

Trace-oblivious type systems. Our type system is trace-oblivious. Liu et al. [18] propose a *memory-trace oblivious* type system for a secure-processor application. In comparison, our program trace also includes *instructions*. Further, Liu et al. propose an indistinguishability-based trace-oblivious notion which is equivalent to a simulation-based notion in their setting. In the secure-computation setting, however, an indistinguishability-based trace-oblivious notion is not equivalent to simulation-based trace obliviousness due to the declassification of computation outputs. We therefore define a simulation-based trace-oblivious notion in our paper which is necessary to ensure the security of the compiled two-party protocol. Other work has proposed type systems that track side channels as traces. For example, Agat's work traces operations in order to avoid timing leaks [3].

III. Technical Overview: Compiling for RAM-Model Secure Computation

Secure RAM-model computation (described in Section II-A), if implemented naively, will be impractical due to reasons described in the remainder of this section. Our key idea is to rely on static analysis to minimize the use of heavyweight cryptographic primitives such as garbled circuits and ORAM.

A. Instruction-Trace Obliviousness

The standard RAM-model secure computation protocol described in Section II-A is relatively inefficient because it requires a secure-computation sub-protocol to compute the universal next-instruction circuit U. This circuit has large size, since it must interpret every possible instruction. In our solution, we will avoid relying on a universal next-instruction circuit, and will instead arrange things so that we can securely evaluate instruction-specific circuits.

Note that it is not secure, in general, to leak information about what instruction is being carried out at each step in the execution of some program. As a simple example, consider a branch over a secret value `s`:

```
if(s) x[i]:=a+b; else x[i]:=a-b
```

Depending on the value of `s`, a different instruction (i.e., `add` or `subtract`) will be executed. To mitigate such an implicit information leak, our compiler transforms a program to an *instruction-trace oblivious* counterpart, i.e., a program whose program-counter value (which determines which instruction will be executed next) does not depend on secret information. The key idea there is to use a **mux** operation to rewrite a secret if-statement. For example, the above code can be re-factored as:

```
O:t1:=s;
O:t2:=a+b;
O:t3:=a-b;
O:t4:=mux(t1, t2, t3);
O:x[i]:=t4
```

At every point during the above computation, the instruction being executed is pre-determined, and so does not leak information about sensitive data. Instruction-trace obliviousness is similar to the program-counter security notion proposed by Molnar et al. [22] (for a different application).

B. Memory-Trace Obliviousness

Using ORAM for memory accesses is also a heavyweight operation in RAM-model secure computation. The naive approach is to place *all* memory in a single ORAM, thus incurring $O(\text{poly}\log n)$ cost per data operation, where n is a bound on the size of the memory.

In the context of securing remote execution against physical attacks, Liu et al. recently observe that not all access patterns of a program are sensitive [18]. For example, a `findmax` program that sequentially scans through an array to find the maximum element has predictable access patterns that do not depend on sensitive inputs. We propose to apply a similar idea to the context of RAM-model secure computation. Our compiler performs static analysis to detect safe memory accesses that do not depend on secret inputs. In this way, we can avoid using ORAM when the access pattern is independent of sensitive inputs. It is also possible to store various subsets of memory (e.g., different arrays) in different ORAMs, when information about which portion of memory (e.g., which array) is being accessed does not depend on sensitive information.

C. Mixed-Mode Execution

We also use static analysis to partition a program into code blocks, and then for each code block use either a public, local, or secure mode of execution (described next). Computation in public or local modes avoids heavyweight secure computation. In the intermediate language, each statement is labeled with its mode of execution.

Public mode. Statements computing on publicly-known variables or variables that have been declassified in the middle of program execution can be performed by both parties independently, without having to resort to a secure-computation protocol. Such statements are labeled `P`. For example, the loop iterators (in lines 1, 3, 10) in Dijkstra's algorithm (see Figure 2) do not depend on secret data, and so each party can independently compute them.

Local mode. For statements computing over Alice's variables, public variables, or previously declassified variables, Alice can perform the computation independently without interacting with Bob (and vice versa). (Here we crucially rely on the fact that we assume semi-honest behavior.) Alice-local statements are labeled `A`, and Bob-local statements are labeled `B`.

Secure mode. All other statements that depend on variables that must be kept secret from both Alice and Bob will be computed using secure computation, making ORAM accesses along the way if necessary. Such statements are labeled `O` (for "oblivious").

D. Example: Dijkstra's Algorithm

In Figure 2, we present a complete compilation example for part of Dijkstra's algorithm. Here one party, Alice, has a private graph represented by a pairwise edge-weight array `e[][]` and the other party, Bob, has a private source/destination pair. Bob wishes to compute the shortest path between his source and destination in Alice's graph.

Our specific implementation of Dijkstra's algorithm uses three arrays, a `dis` array which keeps track of the current shortest distance from the source to any other node; an edge-weight array `orame` which is initialized by Alice's local array `e`, and an indicator array `vis`, denoting whether each node has been visited. In this case, our compiler places arrays `vis` and `e` in separate ORAMs, but does not place array `dis` in ORAM since access to `dis` always follows a sequential pattern.

Note that parts of the algorithm can be computed publicly. For example, all the loop iterators are public values; therefore, loop iterators need not be secret-shared, and each party can independently compute the current loop iteration. The remaining parts of program all require ORAM accesses; therefore, our compiler annotates these instructions to be run in secure mode, and generates equivalent instruction- and memory-trace oblivious target code.

IV. SCVM Language

This section presents SCVM, our language for RAM-model secure computation, along with a type system for this language. We then prove our main results regarding simulatability. In Section IV-A, we present SCVM's formal syntax. In Section IV-B, we give a formal, *ideal world* semantics for SCVM that forms the basis of our security theorem. Informally, each party provides their inputs to an ideal functionality $\mathcal{F}$ that computes the result and returns to each party its result and a trace of events it is allowed to see; these events include instruction fetches, memory accesses, and *declassification* events, which are results computed from both parties' data. Section IV-C includes a formal definition of our security property, which we call Γ-*simulatability*. Informally, a program is secure if each party, starting with its own memory, the program code, and its trace of declassification events, can simulate (in polynomial time) its observed instruction traces and memory traces without knowing the other party's data. We present a type system

```
1 for(i = 0; i < n; ++i) {
2   int bestj = -1; bestdis = -1;
3   for(int j=0; j<n; ++j) {
4     if( ! vis[j] && (bestj < 0
5       || dis[j] < bestdis))
6       bestj = j;
7       bestdis = dis[j];
8   }
9   vis[bestj] = 1;
10   for(int j=0; j<n; ++j) {
11     if( !vis[j] && (bestdis +
12       e[bestj][j] < dis[j]))
13       dis[j] = bestdis + e[bestj][j];
14   }
15 }
```

```
O: orame :=oram( e );
P:i:=0; P:cond1:=i<n;
P:while(cond1) do
    O: bestj :=-1; O: bestdis :=-1;
    P:j:=0; P:cond2:=j<n;
    P:while(cond2) do
        O: t1 := vis [j]; O: t2 :=! t1 ; O: t3 := best <0;
        O: t4 := dis [j]; O: t5 := t4 < bestdis ;
        O: t6 := t3 || t5 ; O: cond3 := t2 && t3 ;
        O: best :=mux( cond3 , j, best );
        O: bestdis :=mux( cond3 , t4 , bestdis );
        P:j:=j+1; P:cond2:=j<n;;
    O: vis [ bestj ]:=1;
    P:j:=0; P:cond2:=j<n;
    P:while(cond2) do
        O: t7 := vis [j]; O: t8 :=! t1 ;
        O: idx := bestj *n; O: idx := idx +j; O: t9 := orame [ idx ];
        O: t10 := bestdis + t9 ; O: t11 := dis [j];
        O: t12 := t10 < t11 ; O: cond4 := t8 && t12
        O: t13 :=mux( cond4 , t10 , t11 ); O: dis [j]:= t13 ;
        P:j:=j+1; P:cond2:=j<n;
```

Fig. 2: Compilation example: Part of Dijkstra's shortest-path algorithm. The code on the left is compiled to the annotated code on the right. Array variable `e` is Alice's local input array containing the graph's edge weights; Bob's input, a source/destination pair, is not used in this part of the algorithm. Array variables `vis` and `orame` are placed in ORAMs. Array variable `dis` is placed in non-oblivious (but secret-shared) memory. (Prior to the shown code, `vis` is initialized to all zeroes except that `vis[source]`—where `source` is Bob's input—is initialized to 1, and `dis[i]` is initialized to `e[source][i]`.) Variables `bestj`, `bestdis` and others in blue background are secret-shared between the two parties.

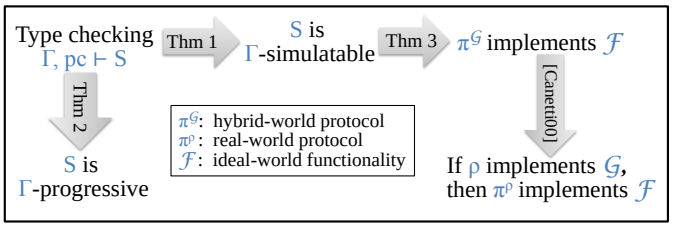


Fig. 3: Formal results.

for SCVM programs in Section IV-D, and in Theorem 1 prove that well-typed programs are Γ-simulatable. Theorem 2 additionally shows that well-typed programs will not get *stuck*, e.g., because one party tries to access memory unavailable to it. Finally, in Section IV-E we define a *hybrid world* functionality that more closely models SCVM's implemented semantics using ORAM, garbled circuits, etc. and prove that for Γ-simulatable programs, the hybrid-world protocol securely implements the ideal functionality. The formal results are summarized in Figure 3.

A. Syntax

In SCVM, each variable and statement has a security label from the lattice {`P`, `A`, `B`, `O`}, where $\sqsubseteq$ is defined to be the smallest partial order such that $\texttt{P} \sqsubseteq l \sqsubseteq \texttt{O}$ for $l \in \{\texttt{A}, \texttt{B}\}$. The label of each variable indicates whether its memory location should be public, known to either Alice or Bob (only), or secret. For readability, we do not distinguish between oblivious secret arrays and non-oblivious secret arrays at this point, and simply assume that all secret arrays are oblivious. Support for non-oblivious, secret arrays can easily be added as discussed in Section V.

An information-flow control type system, which we discuss in Section IV-D, enforces that information can only flow from low (i.e., lower in the partial order) security variables to high security variables. For example, for a statement $x := y$ to be secure, y's security label should be less than or equal to x's security label. An exception is the declassification statement $x := \textbf{declass}_l(y)$ which may declassify a variable y labeled `O` to a variable x with lower security label (depending on l).

The label of each statement indicates the statement's mode of execution. A statement with the label `P` is executed in *public mode*, where both Alice and Bob can see its execution. A statement with the label `A` or `B` is executed in *local mode*, and is visible to only Alice or Bob, respectively. A statement with the label `O` is executed securely, so both Alice and Bob know the statement was executed but do not learn the underlying values that were used.

The syntax of SCVM is given in Figure 4. Most language features are standard. We highlight the statement $x := \textbf{oram}(y)$, by which variable x is assigned to an ORAM initialized with array y's contents, and the expression $\textbf{mux}(x_0, x_1, x_2)$, which evaluates to either x_1 or x_2, depending on whether x_0 is 0 or 1.

Variables	x, y, z	$\in$	**Vars**
Security Labels	l	$\in$	**SecLabels** $= \{\texttt{P}, \texttt{A}, \texttt{B}, \texttt{O}\}$
Numbers	n	$\in$	**Nat**
Operation	op	::=	$+ \mid - \mid ...$
Expressions	e	::=	$x \mid n \mid x\ op\ x \mid$ $x[x] \mid$ **mux**(x, x, x)
Statements	s	::=	**skip** $\mid x := e \mid x[x] := x \mid$ **if** (x) **then** S **else** $S \mid$ **while** (x) **do** $S \mid$ $x :=$ **declass**$_l(y) \mid$ $x :=$ **oram**(y)
Labeled Statements	S	::=	$l : s \mid S; S$

Fig. 4: Syntax of SCVM

B. Semantics

This section defines a formal semantics for SCVM programs. Here we think of this semantics as defining computation carried out, on Alice and Bob's behalf, by an *ideal* functionality $\mathcal{F}$. However, as we foreshadow throughout, the semantics is endowed with sufficient structure that it can be interpreted as using the mechanisms (like ORAM and garbled circuits) described in Sections II and III. We discuss such a *hybrid world* interpretation more carefully in Section IV-E and prove it also satisfies our security properties.

Memories and types. Before we begin, we consider a few auxiliary definitions given in Figure 5. A memory M is a partial map from variables to value-lable pairs. The value is either a natural number n or an array m, which is a partial map from naturals to naturals. The security labels $l \in \{\texttt{P}, \texttt{A}, \texttt{B}, \texttt{O}\}$ indicate the conceptual visibility of the value as described earlier. Note that in a real-world implementation, data labeled `O` is stored in ORAM and secret-shared between Alice and Bob, while other data is stored locally by Alice or Bob. We find it convenient to project from memories the values that are visible at particular labels:

Definition 1 (L-projection)**.** *Given memory M and a set of security labels L, we write $M[L]$ as M's L-*projection*, which is itself a memory such that for all x, $M[L](x) = (v, l)$ if and only if $M(x) = (v, l)$ and $l \in L$.*

We define types **Nat** l and **Array** l, for numbers and arrays, respectively, where l is a security label. A *type environment* Γ associates variables with types, and we interpret it as a partial map. We sometimes consider when a memory is consistent with a type environment Γ:

Definition 2 (Γ-compatibility)**.** *We say a memory M is Γ-*compatible*, if and only if for all x, when $M(x) = (v, l)$, we have $v \in$ **Nat** $\Leftrightarrow \Gamma(x) =$ **Nat** l, and $v \in$ **Array** $\Leftrightarrow \Gamma(x) =$ **Array** l.*

Ideal functionality. Once Alice and Bob have agreed on a program S, we imagine an ideal functionality $\mathcal{F}$ that executes S. Alice and Bob send to $\mathcal{F}$ memories M_A and M_B, respectively. Alice's memory contains data labeled `A` and `P`, while Bob's memory contains data labeled `B` and `P`. (Data labeled `O` is only constructed during execution.) $\mathcal{F}$ then proceeds as follows:

1) It checks that M_A and M_B agree on `P`-labeled values, i.e., that $M_A[\{\texttt{P}\}] = M_B[\{\texttt{P}\}]$. It also checks that they do not share any `A`/`B`-labeled values, i.e., that the domain of $M_A[\{\texttt{A}\}]$ and the domain of $M_B[\{\texttt{B}\}]$ do not intersect. If either of these conditions fail, $\mathcal{F}$ notifies both parties and aborts the execution. Otherwise, it constructs memory M from M_A and M_B:
$$M = \{x \mapsto (v, l) \mid \quad M_A[\{\texttt{A}, \texttt{P}\}](x) = (v, l) \vee M_B[\{\texttt{B}\}](x) = (v, l)\}$$
2) $\mathcal{F}$ executes S according to semantics rules having the form $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'\rangle : D$. This judgment states that starting in memory M, statement S runs, producing a new memory M' and a new statement S' (representing the partially executed program) along with *instruction traces* i_a and i_b, *memory traces* t_a and t_b, and *declassification event* D. We discuss these traces/events shortly. The ideal execution will produce one of three outcomes (or fail to terminate):
   - $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'\rangle : D$, where $D = (d_a, d_b)$. In this case, $\mathcal{F}$ outputs d_a to Alice, and d_b to Bob. Then $\mathcal{F}$ sets M to M' and S to S' and restarts step 2.
   - $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', l : \textbf{skip}\rangle : \epsilon$. In this case, $\mathcal{F}$ notifies both parties that computation finished successfully.
   - $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'\rangle : \epsilon$, where $S' \neq l : \textbf{skip}$, and no rules further reduce $\langle M', S'\rangle$. In this case, $\mathcal{F}$ aborts and notifies both parties.

Notice that the only communication between $\mathcal{F}$ and each party are declassifications d_a and d_b (to Alice and Bob, respectively). This is because we assume that secure programs will always explicitly declassify their final output (and perhaps intermediate outputs, e.g., when processing multiple queries), while all other variables in memory are not of consequence. The memory and instruction traces, though not explicitly communicated by $\mathcal{F}$, will be visible in a real implementation (described later), but we prove that they provide no additional information beyond that provided by the declassification events.

Traces and events. The formal semantics incorporate the concept of traces to define information leakage. There are three types of traces, all given in Figure 5. The first is an instruction trace i. The instruction trace generated by an assignment statement is the statement itself (e.g., $x := e$); the instruction trace generated by a branching statement is denoted **if**(x) or **while**(x). Declassification and ORAM initialization will generate instruction traces **declass**(x, y) and **init**(x, y), respectively. A special instruction trace is ϵ, which means one party cannot observe a trace from a statement's execution (e.g., Bob cannot observe Alice executing her local code). Trace equivalence (i.e. $t_1 \equiv t_2$) is defined inductively in Figure 5.

The second sort of trace is a memory trace t, which captures reads or writes of variables visible to one or the other party. Here are the different memory trace events:

- `P`: Operations on public arrays generate memory event **readarr**(x, n, v) or **writearr**(x, n, v) visible to both parties, including the variable name x, the index n, and the

$$\text{E-Const}\quad l \vdash \langle M, n\rangle \Downarrow_{(\epsilon,\epsilon)} n$$

$$\text{E-Var}\ \frac{M(x) = (v, l') \quad v \in \textbf{Nat} \quad l' \sqsubseteq l \quad (t_a, t_b) = select(l, \textbf{read}(x, v), x)}{l \vdash \langle M, x\rangle \Downarrow_{(t_a,t_b)} v}$$

$$\text{E-Array}\ \frac{M(x) = (m, l') \quad m \in \textbf{Array} \quad l' \sqsubseteq l \quad l \vdash \langle M, y\rangle \Downarrow_{(t'_a,t'_b)} v \quad v' = get(m, v) \quad (t''_a, t''_b) = select(l, \textbf{readarr}(x, v, v'), x) \quad t_a = t'_a @ t''_a \quad t_b = t'_b @ t''_b}{l \vdash \langle M, x[y]\rangle \Downarrow_{(t_a,t_b)} v'}$$

$$\text{E-Op}\ \frac{l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i \quad i = 1, 2 \quad v = v_1\ op\ v_2 \quad t_a = t_{1a} @ t_{2a} \quad t_b = t_{1b} @ t_{2b}}{l \vdash \langle M, x_1\ op\ x_2\rangle \Downarrow_{(t_a,t_b)} v}$$

$$\text{E-Mux}\ \frac{l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i \quad i = 1, 2, 3 \quad v_1 = 0 \Rightarrow v = v_2 \quad v_1 \neq 0 \Rightarrow v = v_3 \quad t_a = t_{1a} @ t_{2a} @ t_{3a} \quad t_b = t_{1b} @ t_{2b} @ t_{3b}}{l \vdash \langle M, \textbf{mux}(x_1, x_2, x_3)\rangle \Downarrow_{(t_a,t_b)} v}$$

Fig. 6: Operational semantics for expressions in SCVM $\boxed{l \vdash \langle M, e\rangle \Downarrow_{(t_a,t_b)} v}$

$$\text{S-Skip}\ \langle M, l : \textbf{skip}; S\rangle \xrightarrow{(\epsilon,\epsilon,\epsilon,\epsilon)} \langle M, S\rangle : \epsilon$$

$$\text{S-Assign}\ \frac{l \vdash \langle M, e\rangle \Downarrow_{(t'_a,t'_b)} v \quad M' = M[x \mapsto (v, l)] \quad (i_a, i_b) = inst(l, x := e) \quad (t''_a, t''_b) = select(l, \textbf{write}(x, v), x) \quad t_a = t'_a @ t''_a \quad t_b = t'_b @ t''_b}{\langle M, l : x := e\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', l : \textbf{skip}\rangle : \epsilon}$$

$$\text{S-Declass}\ \frac{M(y) = (v, \mathtt{0}) \quad l \neq \mathtt{0} \quad (t'_a, t'_b) = select(l, \textbf{write}(x, v), x) \quad t_a = y @ t'_a \quad t_b = y @ t'_b \quad M' = M[x \mapsto (v, l)] \quad i = \mathtt{0} : \textbf{declass}(x, y) \quad D = select(l, (x, v), \epsilon)}{\langle M, \mathtt{0} : x := \textbf{declass}_l(y)\rangle \xrightarrow{(i,t_a,i,t_b)} \langle M', \mathtt{0} : \textbf{skip}\rangle : D}$$

$$\text{S-Cond}\ \frac{(i_a, i_b) = inst(l, \textbf{if}(x)) \quad M(x) = (v, l) \quad (t_a, t_b) = select(l, \textbf{read}(x, v), x) \quad v = 1 \Rightarrow c = 1 \quad v \neq 1 \Rightarrow c = 2}{\langle M, l : \textbf{if}(x)\textbf{then}\ S_1\textbf{else}\ S_2\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M, S_c\rangle : \epsilon}$$

$$\text{S-ORAM}\ \frac{M(y) = (m, l) \quad l \neq \mathtt{0} \quad M' = M[x \mapsto (m, \mathtt{0})] \quad (t'_a, t'_b) = select(l, y, \epsilon) \quad i = \mathtt{0} : \textbf{init}(x, y) \quad t_a = t'_a @ x \quad t_b = t'_b @ x}{\langle M, \mathtt{0} : x := \textbf{oram}(y)\rangle \xrightarrow{(i,t_a,i,t)} \langle M', \mathtt{0} : \textbf{skip}\rangle : \epsilon}$$

$$\text{S-ArrAss}\ \frac{M(y) = (m, l) \quad l \vdash \langle M, x_i\rangle \Downarrow_{(t_{ia},t_{ib})} v_i \quad i = 1, 2 \quad m' = set(m, v_1, v_2) \quad M' = M[y \mapsto (m', l)] \quad (t'_a, t'_b) = select(l, \textbf{writearr}(y, v_1, v_2), y) \quad t_a = t_{1a} @ t_{2a} @ t'_a \quad t_b = t_{1b} @ t_{2b} @ t'_b \quad (i_a, i_b) = inst(l, y[x_1] := x_2)}{\langle M, l : y[x_1] := x_2\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', l : \textbf{skip}\rangle : \epsilon}$$

$$\text{S-While-False}\ \frac{M(x) = (0, l) \quad (i_a, i_b) = inst(l, \textbf{while}(x)) \quad (t_a, t_b) = select(l, \textbf{read}(x, 0), x) \quad S = l : \textbf{while}(x)\textbf{do}\ S'}{\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M, l : \textbf{skip}\rangle : \epsilon}$$

$$\text{S-While-True}\ \frac{M(x) = (v, l) \quad v \neq 0 \quad (t_a, t_b) = select(l, \textbf{read}(x, v), x) \quad S = l : \textbf{while}(x)\textbf{do}\ S'}{\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M, S'; S\rangle : \epsilon}$$

$$\text{S-Seq}\ \frac{\langle M, S_1\rangle \xrightarrow{(i_a,t_a,i_b,i_b)} \langle M', S'_1\rangle : D}{\langle M, S_1; S_2\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'_1; S_2\rangle : D}$$

$$\text{S-Concat}\ \frac{\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,i_b)} \langle M', S'\rangle : \epsilon \quad \langle M', S'\rangle \xrightarrow{(i'_a,t'_a,i'_b,i'_b)} \langle M'', S''\rangle : D}{\langle M, S\rangle \xrightarrow{(i_a @ i'_a, t_a @ t'_a, i_b @ i'_b, t_b @ t'_b)} \langle M'', S''\rangle : D}$$

Fig. 7: Operational semantics for statements in SCVM $\boxed{\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D}$

value read or written v. Operations on public variables generate memory event **read**(x, v) or **write**(x, v).

- `A`: Operations on Alice's secret arrays generate memory event **readarr**(x, n, v) or **writearr**(x, n, v) visible to Alice, but not to Bob. Operations on Alice's secret variables generate memory event **read**(x, v) or **write**(x, v) visible to Alice only.
- `B`: Symmetric to `A`.
- `0`: Operations on a secret array generate memory event x visible to both Alice and Bob, containing only the variable's name, but not the index or the value. A special case is the initialization of ORAM bank x with y's value: a memory trace y, but not its content, is observed.

Memory-trace equivalence is defined in a similar way as instruction-trace equivalence.

Finally, each declassification executed by the program produces a declassification event (d_a, d_b), where Alice learns the declassification d_a and Bob learns d_b. There is also an empty declassification event ϵ, which is used for non-declassification statements. Given a declassification event $D = (d_a, d_b)$, we denote Alice's declassification profile by $D_A = d_a$, and Bob's

$$
\begin{array}{lllll}
\text{Arrays} & m & \in & \textbf{Array} = \textbf{Nat} \rightharpoonup \textbf{Nat} \\
\text{Memory} & M & \in & \textbf{Vars} \rightharpoonup (\textbf{Array} \cup \textbf{Nat}) \times \textbf{SecLabels} \\
\text{Type} & \tau & ::= & \textbf{Nat}\ l \mid \textbf{Array}\ l \\
\text{Type Environment} & \Gamma & ::= & x : \tau \mid \cdot \\
\text{Instruction Traces} & i & ::= & l : x := e \mid l : x[x] := x \mid \\
 & & & l : \textbf{declass}(x, y) \mid l : \textbf{init}(x, y) \\
 & & & l : \textbf{if}(x) \mid l : \textbf{while}(x) \mid i@i \mid \epsilon \\
\text{Memory Traces} & t & ::= & \textbf{read}(x, n) \mid \textbf{readarr}(x, n, n) \mid \\
 & & & \textbf{write}(x, n) \mid \textbf{writearr}(x, n, n) \mid \\
 & & & x \mid t@t \mid \epsilon \\
\text{Declassification} & d & ::= & (x, n) \mid \epsilon \\
\text{Declass. event} & D & ::= & (d, d) \mid \epsilon
\end{array}
$$

$$
select(l, t_1, t_2) = \begin{cases} (t_1, t_1) & \text{if } l = \texttt{P} \\ (t_1, \epsilon) & \text{if } l = \texttt{A} \\ (\epsilon, t_1) & \text{if } l = \texttt{B} \\ (t_2, t_2) & \text{if } l = \texttt{O} \end{cases}
$$

$$
inst(l, i) = select(l, l : i, l : i)
$$

$$
get(m, i) = \begin{cases} m(i) & 0 \le i < |m| \\ 0 & \text{otherwise} \end{cases}
$$

$$
set(m, i, v) = \begin{cases} m[i \mapsto v] & 0 \le i < |m| \\ m & \text{otherwise} \end{cases}
$$

$$
\boxed{t_1 \equiv t_2} \quad t \equiv t \quad t@\epsilon \equiv \epsilon@t \equiv t \quad \frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{t_1@t_2 \equiv t_1'@t_2'}
$$

Fig. 5: Auxiliary syntax and functions for semantics

declassification profile by $D_B = d_b$.

Semantics rules. Now we turn to the semantics, which consists of two judgments. Figure 6 defines rules for the judgment $\langle M, e\rangle \Downarrow_{(t_a,t_b)} v$, which states that under memory M, expression e evaluates to v. This evaluation produces memory trace t_a (resp., t_b) for Alice (resp., Bob). Which memory trace event to emit is chosen using the function $select$, which is defined in Figure 5. One thing worth noticing is the E-Array rule, in which the access to indices out of range is handled by the $get()$ function, which will return a default value 0. Most elements of the rules are otherwise straightforward.

Figure 7 defines rules for the judgment $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$, which says that under memory M, the statement S reduces in one step to memory M' and statement S', while producing instruction trace i_a (resp., i_b) and memory trace t_a (resp., t_b) for Alice (resp., Bob), and generating declassification D. Most rules are standard, except for handling memory traces and instruction traces. Instruction traces are handled using function $inst$ defined in Figure 5. This function is defined such that if the label l of a statement is `A` or `B`, then the other party cannot observe the statement; otherwise, both parties observe the statement.

A skip statement will generate empty instruction traces and memory traces to both parties regardless of its label. An assignment statement will first evaluate the expression to assign, and its trace and the write event constitute the memory trace for this statement.

Declassification $x := \textbf{declass}_l(y)$ results in different views for Alice and Bob. First of all, this rule only allows secret variables to be declassified because it requires the instruction and the declassified variable y to both be labeled `O`, while x's label is *not* `O`. As a result, both parties will observe that y is accessed; they will not necessarily observe its content, which depends (according to the $select$ function) on the security label of x.

ORAM initialization produces a shared, secret array x from an array y provided by one party. The security label of x is thus restricted to be `O`, and the security label of y is restricted not to be `O`. This rule describes that the party possessing y will observe the access to y, and then both parties can observe the access to x.

Rule S-ArrAss handles an array assignment. Similar to rule E-Array, out-of-bounds indices are ignored (cf. the $set()$ function in Figure 5). For if-statements and while-statements, no memory traces will be observed other than those observed from evaluating the guard x.

Rule S-Seq sequences execution of two statements in the obvious way. Finally, rule S-Concat says that if $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M'', S''\rangle : D$, the transformation may perform one or more small-step transformations that generate no declassification.

C. Security

We define the security of a program P using a notion we call Γ-*simulatability*. To state our security theorem, we first define a multi-step version of our statement semantics:

$$
\frac{\begin{array}{c}\langle M, P\rangle \xrightarrow{\Gamma,(i_a,t_a,i_b,t_b)}{}^{\star} \langle M_n, P_n\rangle : D_1, ..., D_n \qquad n \ge 0 \\ \langle M_n, P_n\rangle \xrightarrow{(i_a',t_a',i_b',t_b')} \langle M', P'\rangle : D' \\ D' \neq \epsilon \vee P' = l : \textbf{skip} \qquad M \text{ and } M' \text{ are both } \Gamma\text{-compatible}\end{array}}{\langle M, P\rangle \xrightarrow{\Gamma,(i_a',t_a',i_b',t_b')}{}^{\star} \langle M', P'\rangle : D_1, ..., D_n, D'}
$$

This allows programs to make multiple declassifications, accumulating them as a trace, while remembering only the most recent instruction and memory traces and ensuring that intermediate memories are Γ-compatible.

Definition 3 (Γ-simulatability)**.** *We are given a type environment Γ, and a program P in SCVM, which takes public input, Alice's secret input, and Bob's secret input. We say P is* Γ-simulatable *if there exist simulators sim_A and sim_B such that for all $M, i_a, t_a, i_b, t_b, M', P', D^1, ..., D^n$, if $\langle M, P\rangle \xrightarrow{\Gamma,(i_a,t_a,i_b,t_b)}{}^{\star} \langle M', P'\rangle : D^1, ..., D^n$, then $sim_A(M[\{\texttt{P},\texttt{A}\}], D_A^0, ..., D_A^{n-1}) \equiv (i_a, t_a)$ and $sim_B(M[\{\texttt{P},\texttt{B}\}], D_B^0, ..., D_B^{n-1}) \equiv (i_b, t_b)$.*

Intuitively, this security definition says that if a program P is secure, then there exists a simulator sim_A for Alice such that if the simulator is given public data $M[\{\texttt{P}\}]$, Alice's secret data $M[\{\texttt{A}\}]$, and all outputs $D_A^1, ..., D_A^{n-1}$ declassified to Alice so far, then sim_A can compute the instruction traces i_a and memory traces t_a produced by the ideal semantics up until the next declassification event D^n.

$$\boxed{\Gamma \vdash e : \tau} \qquad \text{T-Var}\frac{\Gamma(x) = \textbf{Nat}\ l}{\Gamma \vdash x : \textbf{Nat}\ l}$$

$$\text{T-Const}\frac{}{\Gamma \vdash n : \textbf{Nat}\ \mathtt{P}}$$

$$\text{T-Op}\frac{\Gamma(x_1) = \textbf{Nat}\ l_1 \quad \Gamma(x_2) = \textbf{Nat}\ l_2}{\Gamma \vdash x_1\ op\ x_2 : \textbf{Nat}\ l_1 \sqcup l_2}$$

$$\text{T-Array}\frac{\Gamma(y) = \textbf{Array}\ l_1 \quad \Gamma(x) = \textbf{Nat}\ l_2 \quad l_2 \sqsubseteq l_1}{\Gamma \vdash y[x] : \textbf{Nat}\ l_1}$$

$$\text{T-Mux}\frac{\Gamma(x_i) = \textbf{Nat}\ l_i \quad i = 1,2,3 \quad l = l_1 \sqcup l_2 \sqcup l_3}{\Gamma \vdash \textbf{mux}(x_1, x_2, x_3) : \textbf{Nat}\ l}$$

$$\boxed{\Gamma, pc \vdash S} \qquad \text{T-Skip}\frac{pc \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow pc = l}{\Gamma, pc \vdash l : \textbf{skip}}$$

$$\text{T-Assign}\frac{\Gamma(x) = \textbf{Nat}\ l \quad \Gamma \vdash e : \textbf{Nat}\ l' \quad pc \sqcup l' \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow l = pc}{\Gamma, pc \vdash l : x := e}$$

$$\text{T-Declass}\frac{pc = \mathtt{P} \quad \Gamma(y) = \textbf{Nat}\ \mathtt{O} \quad \Gamma(x) = \textbf{Nat}\ l \quad l \neq \mathtt{O}}{\Gamma, pc \vdash \mathtt{O} : x := \textbf{declass}_l(y)}$$

$$\text{T-ORAM}\frac{pc = \mathtt{P} \quad \Gamma(x) = \textbf{Array}\ \mathtt{O} \quad \Gamma(y) = \textbf{Array}\ l \quad l \neq \mathtt{O}}{\Gamma, pc \vdash \mathtt{O} : x := \textbf{oram}(y)}$$

$$\text{T-ArrAss}\frac{\Gamma(y) = \textbf{Array}\ l \quad \Gamma(x_1) = \textbf{Nat}\ l_1 \quad \Gamma(x_2) = \textbf{Nat}\ l_2 \quad pc \sqcup l_1 \sqcup l_2 \sqsubseteq l \quad pc \neq \mathtt{P} \Rightarrow l = pc}{\Gamma, pc \vdash l : y[x_1] := x_2}$$

$$\text{T-Cond}\frac{\Gamma(x) = \textbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{O} \quad pc \neq \mathtt{P} \Rightarrow l = pc \quad \Gamma, l \vdash S_i \quad i = 1,2}{\Gamma, pc \vdash l : \textbf{if}(x)\textbf{then}\ S_1\textbf{else}\ S_2}$$

$$\text{T-While}\frac{\Gamma(x) = \textbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{O} \quad pc \neq \mathtt{P} \Rightarrow l = pc \quad \Gamma, l \vdash S}{\Gamma, pc \vdash l : \textbf{while}(x)\textbf{do}\ S}$$

$$\text{T-Seq}\frac{\Gamma, pc \vdash S_1 \quad \Gamma, pc \vdash S_2}{\Gamma, pc \vdash S_1; S_2}$$

Fig. 8: Type System for SCVM

Note that Γ-simulatability is a *termination insensitive* security definition; whether a program terminates or not may leak information [5]. If all runs of a program are sure to terminate (as is typical in secure computation scenarios), then there can be no nontermination leak.

D. Type System

This section presents our security type system, which we prove assures a program's Γ-simulatability (in addition to the standard assurance of *progress*). There are two judgments, both defined in Figure 8. The first is written $\Gamma \vdash e : \tau$, stating that under environment Γ, expression e will evaluate to type τ. The second judgment is written $\Gamma, pc \vdash S$, which states that under environment Γ, and a *label context* pc, a labeled statement S is type-correct. Here, pc is a label that describes the ambient control context; pc will be set according to the guards of enclosing conditionals or loops. Note that since the program cannot execute an if-statement or a while-statement whose guard is secret, pc can be one of `P`, `A`, or `B`, but not `O`. Intuitively, if pc is `A` or `B`, then the statement is Alice's or Bob's local code, respectively. In general, for all labeled statement $S = l : s$, we enforce the invariant $pc \sqsubseteq l$, and if $pc \neq \mathtt{P}$, then $pc = l$. In so doing, if the security label of a statement is `A` (including if-statements and while-statements), then all nested statements also have a security label `A` ensuring they are only visible to Alice. On the other hand, under a public context, the statement label is unrestricted.

Now we consider some interesting aspects of the rules. Rule T-Assign requires $pc \sqcup l' \sqsubseteq l$, as is standard: $pc \sqsubseteq l$ prevents implicit flows, and $l' \sqsubseteq l$ prevents explicit ones. We further restrict that $\Gamma(x) = \textbf{Nat}\ l$, i.e., the assigned variable should have the same security label as the instruction label. Rule T-ArrAss and rule T-Array require that for an array expression $y[x]$, the security label of x should be lower than the security label of y. For example, if x is Alice's secret variable, then y should be either Alice's local array, or an ORAM shared between Alice and Bob. If y is Bob's secret variable, or a public variable, then Bob can observe which indices are accessed, and then infer the value of x. In the Dijkstra example (Figure 2), the array access `vis[bestj]` on line 9 requires that `vis` be an ORAM variable since `bestj` is.

For rules T-Declass and T-ORAM, since declassification and ORAM initialization statements both require secure computation, so we restrict the statement label to be `O`. Since these two statements cannot be executed in Alice's or Bob's local mode, we restrict that $pc = \mathtt{P}$.

Rule T-Cond deals with if-statements; T-While handles while loops similarly. First of all, we restrict $pc \sqsubseteq l$ and $\Gamma(x) = \textbf{Nat}\ l$ for the same reason as above. Further, the rule forbids l to be equal to `O` to avoid an implicit flow revealed by the program's control flow. An alternative way to achieve instruction- and memory- trace obliviousness is through padding [18]. However, in the secure computation scenario, padding achieves the same performance as rewriting a secret-branching statement into a **mux** (or a sequence of them). Further, type using padding would require reasoning about trace patterns, a complication our type system avoids.

We prove that a well-typed program is Γ-simulatable:

Theorem 1. *If $\Gamma, \mathtt{P} \vdash S$, then S is Γ-simulatable.*

Notice that some rules allow program to get stuck. For example, in rule S-ORAM, if the statement is $l : x := \textbf{oram}(y)$, but $l \neq \mathtt{O}$, then the program will not progress. We define a program property, called Γ-*progress*, to formalize the property when a program will not get stuck.

Definition 4 (Γ-progress)**.** *Given a type environment Γ and a program P in SCVM, which takes public input, Alice's secret*

input, and Bob's secret input, we say P is enjoys Γ-progress*, if and only if for any* Γ*-compatible memory* M *such that when* $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D^j$*, for* $j = 0, ..., n-1$*, where* $M_0 = M$*,* $P_0 = P$*, and the* M_j *are* Γ*-compatible, then either* $P_n = l : \textbf{skip}$*, or there exist* $i'_a, t'_a, i'_b, t'_b, M', P'$ *such that* $\langle M_n, P_n\rangle \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M', P'\rangle : D'$.

Theorem 2. *If* $\Gamma, \mathtt{P} \vdash S$*, then* S *enjoys* Γ*-progress.*

Thus, Γ-progress establishes that the third bullet in step (2) of the ideal functionality (Section IV-B) does not happen for type-correct programs.

Proofs of both theorems can be found in Appendix A.

E. Hybrid-world protocol $\pi^{\mathcal{G}}$

Since a straightforward implementation of $\mathcal{F}$ is inefficient, for Γ-simulatable programs, we define a *hybrid-world* protocol $\pi^{\mathcal{G}}$, where $\mathcal{G}$ is a vector of smaller ideal functionalities including binary operations $\mathcal{F}_{op}$, multiplex operation $\mathcal{F}_{\mathtt{mux}}$, ORAM $\mathcal{F}_{\mathtt{oram}}$, and declassification $\mathcal{F}_{\mathtt{declass}}$. The details of these ideal functionalities will be explained in Appendix C. In this protocol, instead of each party delegating the entire computation to $\mathcal{F}$, they proceed in parallel, at times delegating subcomputations to smaller ideal functionalities in $\mathcal{G}$. The final result of this section is Theorem 3, which says that for Γ-simulatable programs, the hybrid-world protocol $\pi^{\mathcal{G}}$ securely implements the ideal functionality $\mathcal{F}$.

Suppose a program is Γ-simulatable, we informally define the hybrid world protocol $\pi^{\mathcal{G}}$. The protocol runs as follows:

1) Alice and Bob set their local declassification lists to empty, i.e. $D_A \leftarrow \emptyset$ and $D_B \leftarrow \emptyset$ and check that their memories agree on public data, i.e., that $M_A[\{P\}] = M_B[\{P\}]$;
2) Alice runs her simulator locally to get $(i_a, t_a) = sim_A(M_A, D_A)$, and Bob run his simulator locally to get $(i_b, t_b) = sim_B(M_B, D_B)$
3) Alice executes the instructions in i_a using data from t_a, and Bob executes the instructions in i_b using data from t_a. Therefore three conditions:
   - If i_a is labeled with `P`, then so is i_b. Both parties execute the statement over their local copies of the public memory.
   - If i_a is labeled with `A`, then Alice executes this instruction locally. So does Bob: if i_b is labeled with `B`, then Bob executes this instruction locally.
   - If i_a is labeled with `O`, then so is i_b. Alice and Bob confirm with each other, and then call the corresponding ideal world functionality to perform the execution.
4) If the last instructions executed in step 3 is a declassification, then go to step 2. Otherwise, stop the protocol.

Theorem 3. *(Informally) If* P *is* Γ*-simulatable, then* $\pi^{\mathcal{G}}$ *emulates* $\mathcal{F}$*. If all SC sub-routines in* $\mathcal{G}$ *are implemented by real world protocols in* ρ *secure against semi-honest adversaries, then the protocol* π^{ρ} *obtained by replacing ideal functionalities in* $\mathcal{G}$ *with corresponding protocols in* ρ *securely computes the functionality* $\mathcal{F}$ *against semi-honest adversaries.*

The formal definition of $\pi^{\mathcal{G}}$, the formal statement of the theorem and the proof of this theorem can be found in Appendix C.

By using the hybrid-world protocol abstraction we keep our security proofs modular, separating the programming language part (Theorems 1 and 2) from the cryptography proofs (Theorem 3). Further, any protocols ρ that securely implement the subroutine ideal functionalities in $\mathcal{G}$ may be plugged in to obtain a real-world protocol. In our evaluation, we use a Garbled Circuit backend to instantiate the real-world protocol.

V. Compilation

We shall informally discuss how to compile a C-like source language with annotation into SCVM programs. An example of our source language is:

```
int sum(alice int x, bob int y) {
  return x<y ? 1 : 0;
}
```

The program's two input variables, x and y, are annotated as Alice's and Bob's data, respectively, while the unannotated return type `int` indicates the result will be known to both Alice and Bob. Programmers need not annotate any local variables. To compile such a program into a SCVM program, the compiler takes the following steps.

Typing the source language. As just mentioned, source level types and initial security label annotations are assumed given. With these, the type checker infers security labels for local variables using a standard security type system [25] using our lattice (Section IV-D). If no such labeling is possible without violating security (e.g., due to a conflict in the initial annotation), the program is rejected.

Labeling statements. The second task is to assign a security label to each statement. For assignment statements and array assignment statements, the label is the least upper bound of all security labels of the variables occurring in the statement. For an if-statement or a while-statement, the label is the least upper bound of all security labels of the guard variables, and all security labels in the branches or loop body.

On secret branching. The type system defined in Section IV-D will reject an if-statement whose guard has security label `O`. As such, if the program branches on secret data, we must compile it into *if-free* SCVM code, using **mux** instructions. The idea is to execute both branches, and use **mux** to activate the relevant effects, based on the guard. To do this, we convert the code into Static-Single-Assignment form (SSA) [4], and then replace occurrences of the ϕ-operator with a **mux**. The following example demonstrates this process:

```
if(s) then x:=1; else x:=2;
```

The SSA form of the above code is

```
if(s) then x1:=1; else x2:=2; x:=phi(x1, x2);
```

Then we eliminate the if-structure and substitute the ϕ-operator to achieve the final code:

```
x1:=1; x2:=2; x:=mux(s, x1, x2)
```

(Note that, for simplicity, we have omitted the security labels on the statements in the example.)

On secret while loops. The type system requires that while loop guards only reference public data, so that the number of iterations does not leak information. A programmer can work around this restriction by imposing a constant bound on the loop; e.g., manually translating `while (s) do` S to `while (p) do if (s)` S `else skip`, where `p` defines an upper bound on the number of iterations.

Declassification. The compiler will emit a declassification statement for each return statement in the source program. To avoid declassifying in the middle of local code, the type checker in the first phase will check for this possibility and relabel statements accordingly.

Extension for non-oblivious secret RAM. The discussion so far supports only secret ORAMs. To support non-oblivious secret RAM in SCVM, we shall add an additional security label `N` such that $\texttt{P} \sqsubseteq \texttt{N} \sqsubseteq \texttt{O}$. To incorporate such a change, the memory trace for the semantics should include two more kinds of trace event, **nread**(x, i) and **nwrite**(x, i), which represent that only the index of an access is leaked, but not the content. Since label `N` only applies to arrays, we allow types **Array** `N` but not types **Nat** `N`. The rules T-Array and T-ArrAss should be revised to deal with the non-oblivious RAM. For example, for rule T-ArrAss, where l_1 is the security label for the array, l_2 is the security label of the index variable and l_3 is the security label of the value variable, the type system should still restrict $l_3 \sqsubseteq l_1$, but if $l_1 = \texttt{N}$, the type system shall accept $l_2 = \texttt{O}$.

Correctness. We do not prove the correctness of our compiler, but instead use our SCVM type checker (using the above extension) for the generated SCVM code, ensuring it is Γ-simulatable. Ensuring the correctness of compilers is orthogonal and outside the scope of this work, and existing techniques [1] can potentially be adapted to our setting.

Compiling Dijkstra's algorithm. We explain how compilation works for Dijkstra's algorithm, previously shown in Figure 2. First, the type checker for the source program determines how memory should be labeled. It determines that the security labels for `bestj` and `bestdis` should be `O`, and the arrays `dis` and `vis` should be secret-shared between Alice and Bob, since their values depend on both Alice's input (i.e. the graph's edge weights) and Bob's input (i.e. the source). Then, since on line 9 array `vis` is indexed with `bestj`, variable `vis` should also be put in an ORAM. Similarly, on line 12, array `e` is indexed by `bestj` so it must also be secret; as such we must promote `e`, owned by Alice, to be in ORAM, which we do by initializing a new ORAM-allocated variable `orame` to `e` at the start of the program.

The type checker then uses the variable labeling to determine the statement labeling. Statements on lines 4–7, 9, and 11–13, require secure computation and thus are labeled as `O`. Loop control-flow statements are computed publicly, so they are labeled as `P`.

The two if-statements both branch on ORAM-allocated data, so they must be converted to **mux** operations. Lines 4–7 are transformed (in source-level syntax) as follows

```
cond2:=!vis[j]&&(bestj<0||dis[j]<bestdis);
bestj := mux(cond2, j, bestj);
bestdis := mux(cond2, dis[j], bestdis);
```

Lines 11-13 are similarly transformed

```
tmp:=bestdis+orame[bestj*n+j];
cond3:=!vis[j]&&(tmp<dis[j]);
dis[j]:=mux(cond3, tmp, dis[j]);
```

Finally, the code are translated into SCVM's three-address code style syntax.

VI. Evaluation

Programs. We have built several secure two-party computation applications. For run-once tasks, we considered Knuth-Morris-Pratt (KMP) string matching and Dijkstra shortest distance algorithm. Both algorithms have interesting security applications involving more than one party providing their private data, such as KMP for intrusion detection and Dijkstra for social network mining, but mostly relevant for large datasets. Thus, it is important to construct efficient secure computation protocols for them. For repeated sublinear-time database queries, we considered binary search and the oblivious heap data structure. The programs we will discuss are listed in Table II.

Compilation time. All programs took unnoticeable time for compilation, i.e., *under* 1 *second*. Therefore, we do not separately report the compile time for each program. In comparison, some earlier circuit-model compilers involve copying datasets into circuits constructed, and therefore the compile-time can be large [16], [20] (e.g., Kreuter et al. report roughly 1000s compile time for 16-node graph isomorphism [16]).

In our experiments, we manually checked the correctness of compiled programs. To automate this step, orthogonal work on automatically ensuring compiler correctness [1] can be adapted to our setting.

A. Evaluation Methodology

Although our techniques are compatible with any cryptographic backend (secure in the semi-honest model), we use the garbled circuit approach in our evaluation. Specifically, we use the cryptographic backend described by Huang et al. [13].

All applications are tested in the semi-honest setting. We measure the costs by calculating the number of encryptions required by the party running as the circuit generator (the party running as the evaluator will have less work due to the point-and-permute optimization [23]). That is, for every non-free binary gate, the generator needs to run 3 symmetric cipher operations, and for every OT, 2 symmetric cipher operations are required using the OT extension technique by Ishai et al. [14].

TABLE II: Programs used in our evaluation

Name	Alice's Input	Bob's Input	Setting
Dijkstra shortest distance	a graph	a (src, dest) pair	run-once
Knuth-Morris-Pratt matching	a sequence	a pattern	run-once
Aggregation over stream	a key-value table	a stream	run-once
Inverse permutation	share of permutation	share of permutation	run-once
Binary search	sorted array	search key	repeated sublinear-time query
Heap (insertion/extraction)	share of the heap	share of the heap	repeated sublinear-time query

We implemented the tree-based ORAM by Shi et al. [26] completely using garbled circuits, so that array accesses reveal nothing about the addresses nor the outcomes. Following the ORAM encryption technique proposed in [10], every block is xor-shared while the client's share is always an output of client's secret cipher. This adds 1 additional cipher operation per block (when the length of an ORAM block is less than the width of the cipher). We note specific choices of the ORAM parameters in related discussion of each application.

Metrics. We use the number of symmetric encryptions (128-bit AES) as our performance metrics. Measuring the performance by the number of symmetric encryptions (instead of wall clock times) makes it easier to compare with other systems since the numbers can be independent of the underlying hardware and ciphering algorithms. Additionally, in our experiments these encryption numbers also represent the bandwidth consumption since it happens to be the case that every encryption will be sent over the network. Therefore, we do not report separately the bandwidth used. Modern processors with AES extensions can compute AES encryptions at 10^8 encryptions per second [2].

B. Comparison with Automated Circuits

Presently, automated secure computation largely focus on the circuit-model of computation, and they all handled array accesses by linearly scanning the entire array with a circuit every time an array lookup happens — this incurs prohibitive overhead when the dataset is large. In this section, we mainly compare our approach with the existing compiled circuits, and demonstrate that our approach scales much better with respect to dataset size.

1) Repeated sublinear-time queries: Recall that it always requires a linear (in the size of the RAM) amount of work to initialize the RAM for secure computation. However, because the ORAM initialization can be amortized over multiple queries, we can achieve sublinear amortized cost for sublinear-time, repeated queries.

Binary search. One example execution we tested in our approach is binary search, where one party owns a confidential sequence of data (sorted) and the other party tries to look up with secret index.

In all experiments of this paper, we set the ORAM bucket size to 32 (i.e., each tree-node can store up to 32 blocks). For binary search, we aligned our experiment settings with that of Gordon et al. [10], namely, assuming the size of each data item is 512 bits. We set the recursion factor to 8 (i.e., each block can store upto 8 indices for the data in the upper level recursion tree) and the recursion cut-off threshold to 1000 (namely no more recursion once less than 1000 units are to be stored). Comparing to a circuit-model compiled implementation where every read scans the whole memory, our approach is faster for all RAM sizes being tested (Figure 9(a)). On data of size 2^{20}, our approach achieves a $100\times$ speedup.

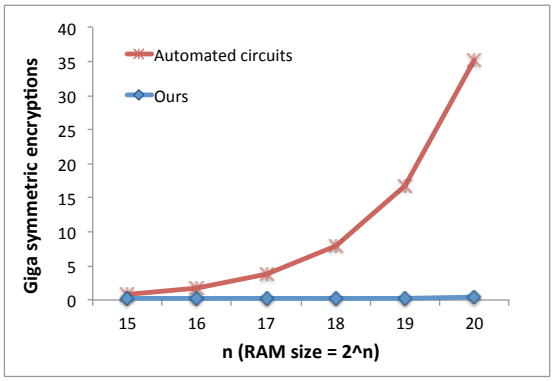
Admittedly, a simpler one-time linear scan suffices solving the search problem, but such solution uses external human insights that are in general difficult to infer from the original input program (say, the input program was written following idea of binary search). However, even compared to such an implementation, our approach still runs faster when the size of the dataset is greater than 256K (Figure 9(b)). On data of size 2^{20}, our approach achieves a $5\times$ speedup.

Heap. Besides binary search, we also implemented an oblivious heap data structure (with 32-bit payload, i.e., size of each item). The costs of insertion and extraction respecting various heap sizes are given in Figure 10(a) and 10(b), respectively. The basic shapes of the performance curves are very similar to that for binary search (except that heap extraction runs twice slower than insertion because two comparisons are needed per level). We can observe an $18\times$ speedup for both heap insertion and heap extraction when the heap size is 2^{20}.

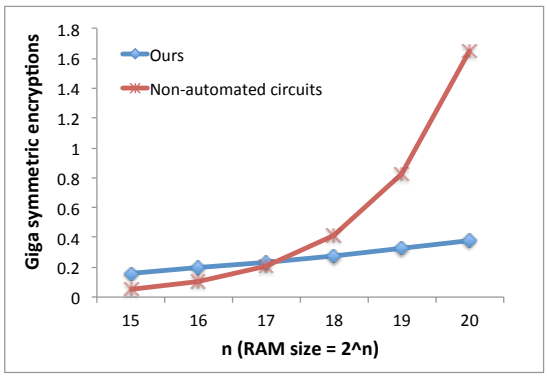
The speedup of our heap implementation over automated circuits is even greater when the size of the payload is bigger. At 512-bit payload, we have an 100x speedup for data size 2^{20}. This is due to the extra work incurred from realizing the ORAM mechanism, which grows (in poly-logarithmic scale) with the size of the RAM but independent of the size of each data item.

2) Faster one-time executions: We present two applications: Knuth-Morris-Pratt sequence matching (representative of linear time RAM programs) and Dijkstra shortest distances calculation (representative of super-linear time RAM programs). Both algorithms are known to be asymptotically optimal assuming the RAM model of computation. However, it is unclear how to construct linear size circuits to do sequence matching (or n-square size circuits to compute graph shortest distances, respectively).

Knuth-Morris-Pratt matching. Alice has a secret string T (of length n) while Bob has a secret pattern P (of length m) and wants to scan through Alice's string looking for this pattern. The original KMP algorithm runs in $O(n+m)$ time when T and P are in plaintext. Our compiler compiles an implementation of KMP into a secure string matching protocol preserving its linear efficiency upto an polylogarithmic factor due to the ORAM technique.

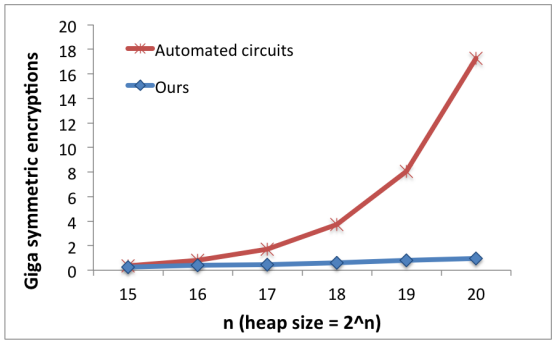


(a) Our approach vs. automated circuit-based approach

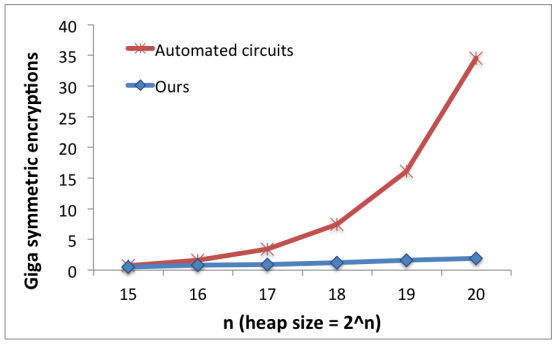


(b) Our approach vs. hand-constructed linear scan circuit

Fig. 9: Binary search



(a) Heap insertion



(b) Heap extraction

Fig. 10: Heap operations

We assume the string T and the pattern P both consist of 16-bit characters. The recursion factor of the ORAM is set to 16. Figure 11(a) and 11(b) show our results compared to those when a circuit-model compiler is used. From Figure 11(a), we can observe that our approach is slower than the circuit-based approach on small datasets, since the overhead of the ORAM protocol dominants in such cases. However, the circuit-based approach's runtime increases much faster than our approach as long as the dataset's size increases. When $m = 50$ and $n = 2 \times 10^6$, our program runs $21\times$ faster.

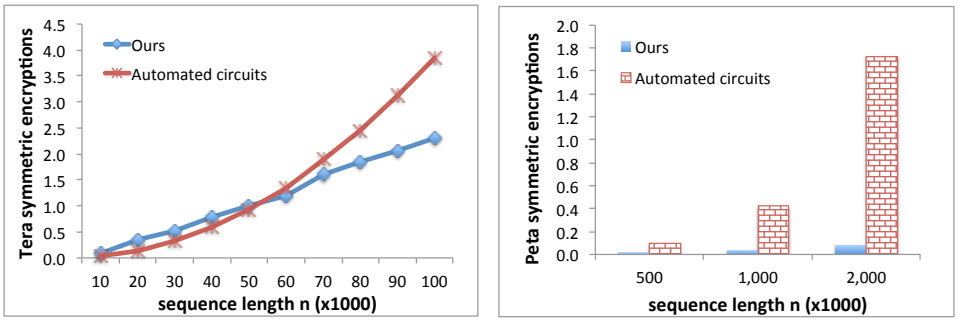
Dijkstra shortest distances. Alice has a secret graph while Bob has a secret source, destination pair, and wishes to compute the shortest distance between them (e.g., a Google Map navigation scenario). Our compilation product is a $O(n^2 \log^3(n))$ (in terms of the number of encryptions) protocol compiled from the standard Dijkstra shortest distance algorithm.

In our experiment, Alice's graph is represented by an $n \times n$ adjacency matrix (of 32-bit integers) where n is the number of vertices in the graph. The distances associated with the edges are denoted by 32-bit integers. We set ORAM recursion factor to 8. The results (Figure 12(a)) show that our scheme runs faster for all sizes of graphs tested. As the performance of our protocol is barely noticeable in Figure 12(a), the performance gaps between the two protocols for various n is explicitly plotted in Figure 12(b). Note the shape of the speedup curve is roughly quadratic.

Aggregation over a stream. Consider a typical stream database scenario where Alice provides a key-value table, Bob provides a stream stored in an array. The secure computation keeps a window of data from Bob's stream, looks up the values for all keys in the window, and compute the minimal of all values in the window. Our compiler outputs a $O(nlog^3 n)$ protocol to accomplish the task. The optimized protocol performs significantly better, as shown in Figure 13 (we fixed the window size k to 1000 and set recursion factor to 8, while varying the dataset from 1 to 6 million pairs).

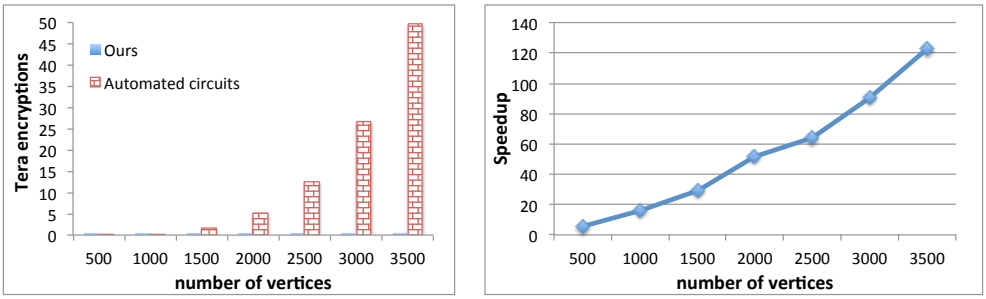
C. Comparison with RAM-SC Baselines

Benefits of instruction-trace obliviousness. The RAM-SC theory by Gordon et al. [10] uses a universal next-instruction circuit to implement a CPU for the purpose of hiding the program counter and which instructions have been executed. This requires every instruction to incur ORAM operations (for instruction and data fetches). Further, since the next-instruction must interpret all instructions, the circuit must effectively execute of all possible instructions and use a mux to select the right outcome.



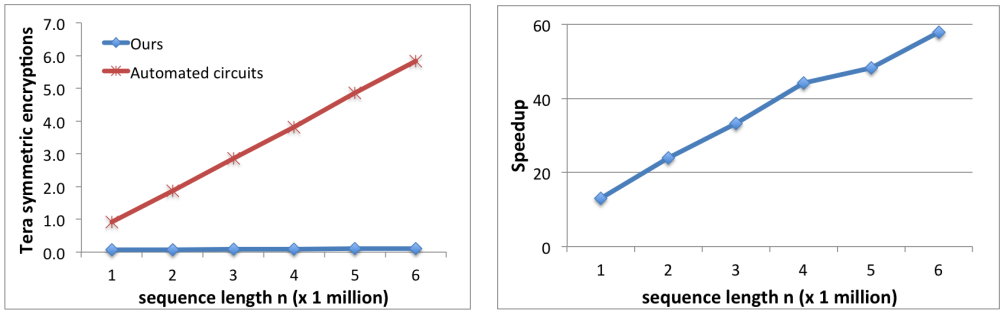
(a) Median n (fixing $m = 50$) (b) Large n (fixing $m = 50$)

Fig. 11: KMP string matching



(a) Performance (b) Speedup

Fig. 12: Dijkstra shortest distances



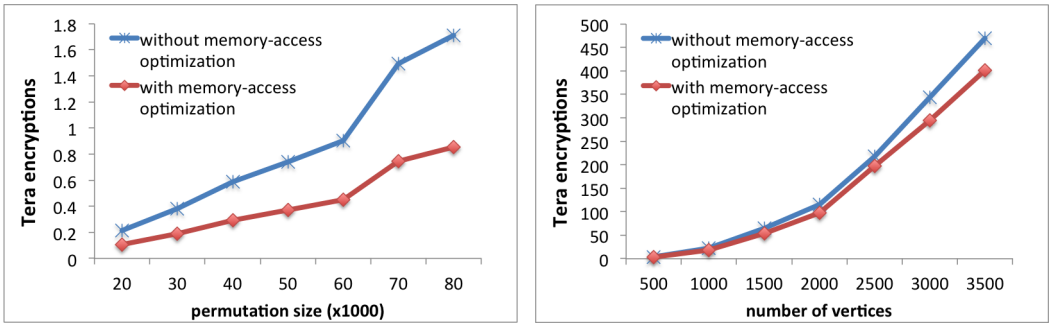
(a) Performance (b) Speedup

Fig. 13: Aggregation over stream

Although theoretical constructions using the next-instruction circuit based approach were known, concrete implementations remains unavailable. We use simple back-of-envelop calculations to show that our approach would be orders-of-magnitude faster.

Consider the problem of binary search over a 1-million item dataset: in each iteration, there is roughly 10 instructions to run, hence 200 instructions in total to complete the search. To run every instruction, a universal-circuit-based CPU implementation has to execute every possible instruction defined in its instruction set. Even if we conservatively assume a RISC-style CPU design, it amounts to over 9 million (non-free) binary gates to execute just a memory read/write over a 512M bit RAM. Plus, an extra ORAM read is required to obliviously fetch every instruction. Thus, at least a total of 3600 million binary gates are needed, which is more than 20 times slower than our result exploiting instruction trace obliviousness. Furthermore, notice that binary search is merely a case where the program traces are very short (with only logarithmic length). Due to the overwhelming cost of ORAM read/write instructions, we stress that the performance gap will be much greater with respect to programs that have relatively fewer memory read/write instructions (comparing to binary search, 1 out of 10 instructions is a memory read instruction).

Benefits of memory-trace obliviousness. Figure 14 demonstrates the savings due to the memory-trace oblivious optimiza-



(a) Inverse permutation (b) Dijkstra

Fig. 14: Savings by memory-trace obliviousness optimization. In (a), the non-linearity (around 60) of the curve is due to the increase of the ORAM recursion level at that point.

tion in two applications.

- **Inverse permutation.** Consider a permutation of size n, represented by an array `a` of n distinct numbers from 1 to n, i.e., the permutation maps the `i`-th object to the `a[i]`-th object. One common computation would be to compute its inverse, e.g., to do an inverse table lookup using secret indices. The inverse permutation (with result stored in array `b`) can be computed with the loop:
  `while (i < n) { b[a[i]]=i; i=i+1;}`
  The memory-trace obliviousness optimization automatically identifies that the array `a` doesn't need to be put in ORAM even its content should remain secret (because the access pattern to `a` is entirely public known). This yields 50% savings, which is corroborated by our experiment results (Figure 14(a)).
- **Dijkstra.** Our second example of the savings from memory trace analysis was given in Section III, when discussing the Dijkstra algorithm. Our experiments show that we consistently saved $15 \sim 20\%$ for all graph sizes. The savings rates for smaller graphs are in fact higher even though it is barely noticeable in the chart because of the fast (super-quadratic) growth of overall cost.

VII. Conclusion

We describe the first automated approach towards efficient RAM-model secure computation, fundamentally addressing inherent limitations of the circuit model prevalently adopted, and offering opportunities to scale up secure computation to big data sizes. Directions for future work include extending our framework to support malicious security; applying orthogonal techniques [1] to ensure the compiler correctness; incorporating other cryptographic backends into our framework; and adding additional language features such as high-dimensional arrays and structured data types.

Acknowledgments. We thank Hubert Chan, Dov Gordon, Feng-Hao Liu, Emil Stefanov, and Hong-Sheng Zhou for helpful discussions. We also thank the anonymous reviewers and our shepherd for their insightful feedback and comments. This research was funded by NSF awards CNS-1111599, CNS-1223623, and CNS-1314857, a Google Faculty Research Award, and by the US Army Research Laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] COMPCERT. http://compcert.inria.fr/.
[2] Hardware aes showdown - via padlock vs intel aes-ni vs amd hexacore. http://grantmcwilliams.com/tech/technology/item/532-hardware-aes-showdown-via-padlock-vs-intel-aes-ni-vs-amd-hexacore.
[3] J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000.
[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.
[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.
[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Security & Privacy*, 2013.
[7] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Usenix Security Symposium*, 2013.
[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3), May 1996.
[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.
[10] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.
[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, 2010.
[12] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *CCS*, 2012.
[13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-

party computation using garbled circuits. In *Usenix Security Symposium*, 2011.
[14] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, 2003.
[15] F. Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.
[16] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security Symposium*, 2013.
[17] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.
[18] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *CSF*, 2013.
[19] S. Lu and R. Ostrovsky. How to garble ram programs. In *EUROCRYPT*, 2013.
[20] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: a secure two-party computation system. In *USENIX Security Symposium*, 2004.
[21] P. Mardziel, M. Hicks, J. Katz, and M. Srivatsa. Knowledge-oriented secure multiparty computation. In *PLAS*, 2012.
[22] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005.
[23] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, 2009.
[24] A. Rastogi, P. Mardziel, M. Hammer, and M. Hicks. Knowledge inference for optimizing secure multi-party computation. In *PLAS*, 2013.
[25] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.
[26] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.
[27] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
[28] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.
[29] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.
[30] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. *IEEE Symposium on Security and Privacy*, pages 493–507, 2013.
[31] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *S & P*, 2013.

APPENDIX

A. Proof of Theorem 1

We begin by discussing how to construct sim_A; the simulator sim_B is constructed similarly.

The intuition is to write a semantics that sim_A can follow to generate instruction traces, memory traces and memories. The challenge is that sim_A may not observe the full memory. Therefore, we shall first create a profile of the memory from sim_A's point of view. Particularly, we first define the Alice-similarity property as follows:

Definition 5. *We say two memories M_1 and M_2 are Alice-similar, denoted as $M_1 \sim_A M_2$, if and only if $\forall x. M_1(x) = (v, l) \wedge l \sqsubseteq \mathtt{A} \Leftrightarrow M_2(x) = (v, l) \wedge l \sqsubseteq \mathtt{A}$.*

Since Alice does not have the view of Bob's local data, and those data secret-shared between them two, we define a special notion E as the values not observable to Alice. We define the operations on top of E as follows:

$$\mathrm{E}\ op\ v = \mathrm{E} \quad v\ op\ \mathrm{E} = \mathrm{E} \quad \mathrm{E}(v) = \mathrm{E} \quad m(\mathrm{E}) = \mathrm{E}$$

We define the following auxiliary functions accordingly:

$$\begin{array}{rcl} (select_A(l,t,t'), select_B(l,t,t')) & := & select(l,t,t') \\ read_A(l,v) & := & \begin{cases} v & l \sqsubseteq \mathtt{A} \\ \mathrm{E} & \text{otherwise} \end{cases} \\ val(v,l) & := & v \\ val(m,l) & := & m \end{array}$$

Then we define the semantics for sim_A in Figure 15 to evaluate an expression over the memory profile. The following lemma shows that the semantics for sim_A actually generates the same memory trace as the semantics for SCVM.

Lemma 1. *If $\langle M, e\rangle \Downarrow_{(t_a,t_b)} v$, $\Gamma \vdash e : l'$, $\Gamma, l \vdash \langle M', e\rangle \Downarrow_t v'$, $M \sim_A M'$, and $l' \sqsubseteq l$, then $t_a \equiv t$ and if $l \sqsubseteq \mathtt{A}$, then $v = v'$, otherwise $v' = \mathrm{E}$.*

Proof: Prove by structural induction on e. If $e = x$, then $\Gamma(x) = \mathbf{Nat}\ l'$. If $l \sqsubseteq \mathtt{A}$, then $v' = val(M'(x)) = val(M(x)) = v$, therefore $v = v'$. Further $t = \mathbf{read}(x, v') = \mathbf{read}(x, v) = t_a$ if $l \sqsubseteq \mathtt{A}$. If $l = \mathtt{B}$, then $v' = \mathrm{E}$, and $t = \epsilon = t_a$. If $l = \mathtt{0}$, then $v' = \mathrm{E}$, and $t = x = t_a$.

If $e = n$, then $t = \epsilon = t_a$, and $v' = n = v$, and $l = \mathtt{P} \sqsubseteq \mathtt{A}$.

If $e = x_1\ op\ x_2$. Then we know $\Gamma \vdash \langle M', x_i\rangle \Downarrow_{t_i} v_i'$, and $\langle M, x_i\rangle \Downarrow_{(t_a^i, t_b^i)} v_i$ for $i = 1, 2$. By induction assumption, we know $t_i \equiv t_a^i$, and thus $t = t_1 @ t_2 \equiv t_a^1 @ t_a^2 = t_a$. For value, suppose $\Gamma(x_i) = \mathbf{Nat}\ l_i$, $i = 1, 2$, if $l \sqsubseteq \mathtt{A}$, then $l_i \sqsubseteq \mathtt{A}$ holds true, and by induction assumption, we know $v_i = v_i'$ for $i = 1, 2$, and thus $v = v_1\ op\ v_2 = v_1'\ op\ v_2 = v'$. Otherwise, we know $v' = \mathrm{E}$.

If $e = x[y]$. We first reason about the value. If $l \sqsubseteq \mathtt{A}$, then suppose $\Gamma(y) = \mathbf{Nat}\ l''$, then $l'' \sqsubseteq l' \sqsubseteq l \sqsubseteq \mathtt{A}$ according to $\Gamma \vdash x[y] : l'$. Then we know $v_1' = val(M'(y)) = val(M(y)) = v_1$. Further, we know $(m', l) = M'(x) = M(x) = (m, l)$, and thus $v' = get(m', v_1') = get(m, v_1) = v$. If $l \not\sqsubseteq \mathtt{A}$, then $v = \mathrm{E}$.

Then we reason about the trace. If $l \sqsubseteq \mathtt{A}$, then $t = \mathbf{read}(y, v_1) @ \mathbf{readarr}(x, v_1, v) \equiv \mathbf{read}(y, v_1') @ \mathbf{readarr}(x, v_1, v') = t_a$. If $l = \mathtt{B}$, we have $t \equiv \epsilon \equiv t_a$. If $l = \mathtt{0}$, we have $t \equiv y @ x \equiv t_a$.

For $e = \mathbf{mux}(x_1, x_2, x_3)$, based on a very similar argument as for $x_1\ op\ x_2$, we can get the conclusion. ∎

The following lemma further claims that if an expression has a type B, then it will generate no observable instruction traces and memory traces.

Lemma 2. *If $\Gamma \vdash e : l'$, and $\Gamma, \mathtt{B} \vdash \langle M, e\rangle \Downarrow_t v$, then $t \equiv \epsilon$.*

Proof: Prove by structure induction on e. If $e = x$, then $t = \epsilon$ by rule Sim-E-Var.

If $e = x_1\ op\ x_2$. Suppose $\Gamma \vdash x_i : l_i$ for $i = 1, 2$, then we know $l_i \sqsubseteq \mathtt{B}$. Therefore $t_i \equiv \epsilon$, and thus $t \equiv \epsilon$.

$$\boxed{\Gamma \vdash \langle M, e\rangle \Downarrow_{t_a)} v}$$

$$\text{Sim-E-Const}\quad \Gamma, l \vdash \langle M, n\rangle \Downarrow_{\epsilon} n$$

$$\text{Sim-E-Var}\ \frac{\Gamma(x) = \textbf{Nat}\ l' \qquad v = read_A(l, val(M(x)))\qquad t = select_A(l, \textbf{read}(x, v), x)}{\Gamma, l \vdash \langle M, x\rangle \Downarrow_t v}$$

$$\text{Sim-E-Op}\ \frac{\Gamma(x_i) = \textbf{Nat}\ l_i \qquad \Gamma, l \vdash \langle M, x_i\rangle \Downarrow_{t_i} v_i \qquad t = t_1@t_2 \qquad v = v_1\ op\ v_2 \qquad i = 1, 2}{\Gamma, l \vdash \langle M, x_1\ op\ x_2\rangle \Downarrow_t v}$$

$$\text{Sim-E-Array}\ \frac{\begin{array}{c}\Gamma(y) = \textbf{Nat}\ l' \qquad \Gamma, l \vdash \langle M, y\rangle \Downarrow_{t_1} v_1 \\ \Gamma(x) = \textbf{Array}\ l'' \qquad v_1 = val(M(y)) \\ t_2 = select_A(l, \textbf{readarr}(x, v_1, v), x) \\ t = t_1@t_2 \qquad v = read_A(l, val(M(x))(v_1))\end{array}}{\Gamma, l \vdash \langle M, x[y]\rangle \Downarrow_t v}$$

$$\text{Sim-E-Mux}\ \frac{\begin{array}{c}\Gamma(x_i) = \textbf{Nat}\ l_i \qquad \Gamma, l \vdash \langle M, x_i\rangle \Downarrow_{t_i} v_i \qquad i = 1, 2, 3 \\ v_1 \neq 0 \Rightarrow v = v_3 \qquad v_1 \neq 1 \Rightarrow v = v_2 \\ v_1 = \mathtt{E} \Rightarrow v = \mathtt{E} \qquad t = t_1@t_2@t_3\end{array}}{\Gamma, l \vdash \langle M, \textbf{mux}(x_1\, x_2, x_3)\rangle \Downarrow_t v}$$

Fig. 15: Operational semantics for sim_A

If $e = x[y]$, the conclusion follows the fact that $\Gamma, \mathtt{B} \vdash \langle M, y\rangle \Downarrow_\epsilon v$, and $select_A(\mathtt{B}, \textbf{readarr}(x, v_1, v), x) = \epsilon$.

If $e = \textbf{mux}(x_1, x_2, x_3)$, similar to binary operation, we know $t \equiv \epsilon$. ∎

The following lemma is the main lemma saying that evaluating Alice-similar memories, sim_A and SCVM will generate the same instruction traces and memory traces, and results in Alice-similar memory profiles.

Lemma 3. *If* $\langle M, S\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M^\star, S'\rangle : D$*, where* $D \neq \epsilon$*,* $\Gamma, \mathtt{P} \vdash S$*,* $M \sim_A M'$*, and* $\langle M', S, D\rangle \xrightarrow{(i,t)} \langle M^{\star\star}, S''\rangle$*, then* $S = S''$*,* $M^\star \sim_A M^{\star\star}$*,* $i_a \equiv i$ *and* $t_a \equiv t$*.*

Proof: The conclusion $S' = S''$ can be trivially done by examining the correspondence of each E- and S- rules and Sim- rules. Therefore, we only prove (1) $M^\star \sim_A M^{\star\star}$, (2) $i_a \equiv i$, and (3) $t_a \equiv t$.

We prove by induction on the length of steps L toward generating declassification event D. If $L = 0$, then we know $S = l : x := \textbf{declass}_l(y); S_2$ (or $\mathtt{O} : x := \textbf{declass}_l(y)$. By typing rule T-Declass, $l \neq \mathtt{O}$, $\Gamma(x) = \textbf{Nat}\ l$. If $l \sqsubseteq \mathtt{A}$, then $D_A = (x, v)$, and thus $M^{\star\star} = M'[x \mapsto v] \sim_A M[x \mapsto v] = M^\star$. Further, we know $i_a = \textbf{declass}(x, y) = i$, and $t_a = y@\textbf{write}(x, v) = t$. Second, if $l_x = \mathbf{B}$, then $M^{\star\star} = M' \sim_A M = M^\star$, $i_a = \textbf{declass}(x, y) = i$, and $t_a = y = t$.

We next consider $L > 0$, then $S = S_1; S_2$. Since $(S_a; S_b); S_c$ is equivalent to $S_a; (S_b; S_c)$ in the sense that if $\langle M, (S_a; S_b); S_c\rangle \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M', S'\rangle : D$, then $\langle M, S_a; (S_b; S_c)\rangle \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M', S'\rangle : D$, where $i_a \equiv i'_a$, $i_b \equiv i'_b$, $t_a \equiv t'_a$, and $t_b \equiv t'_b$. Therefore we only consider S_1 not to be a Seq statement, then we know $S_1 = l : s_1$. By taking one step, we only need to prove claims (1)-(3), then the conclusion can be shown by induction assumption. In the following, we consider how this step is executed.

Case l : **skip.** If $S_1 = l :$ **skip**, the conclusion is trivial, i.e. $i_a = \epsilon = i$ and $t_a = \epsilon = t$ and $M^{\star\star} = M' \sim_A M = M^\star$.

Case $l : x := e$**.** If $S_1 = l : x := e$, $i_a = l : x := e = i$. Then we show $t \equiv t_a$. If $l \sqsubseteq \mathtt{A}$, $t \equiv t_a$ directly follows Lemma 1. If $l = \mathtt{B}$, then by Lemma 2, we have $t \equiv \epsilon \equiv t_b$. If $l = \mathtt{O}$, then we consider e separately. If $e = y$, then $t = y@x = t_a$. If $e = y[z]$, then $t = z@y@x = t_a$. If $e = n$, then $t = x = t_a$. If $e = y\ op\ z$, then $t = y@z@x = t_a$. Finally, if $e = \textbf{mux}(x_1, x_2, x_3)$, then $t = x_1@x_2@x_3@x = t_a$.

Finally, we prove the memory equivalence. If $l \sqsubseteq \mathtt{A}$, then according to Lemma 1, e evaluates to the same value v in the semantics, and in the simulator. Therefore $M^{\star\star} = M'[x \mapsto v] \sim_A M[x \mapsto v] = M^\star$. If $\mathtt{B} \sqsubseteq l$, then $M^{\star\star} = M' \sim_A M \sim_A M[x \mapsto v] = M^\star$. Therefore, the conclusion is true.

Case $\mathtt{O} : x := \textbf{oram}(y)$**.** It is easy to see that $M^{\star\star} = M' \sim_A M \sim_A M[x \mapsto m] = M^\star$, and $i = \mathtt{O} : \textbf{init}(x, y) = i_a$. Suppose $\Gamma(y) = \textbf{Nat}\ l$, then we know $l \neq \mathtt{O}$. If $l \sqsubseteq \mathtt{A}$, then $t = y@x = t_a$. Otherwise, $l = \mathtt{B}$, then we know $t = x = t_a$.

Case $l : y[x_1] := x_2$**.** By typing rule T-ArrAss, we know $\Gamma(y) = \textbf{Array}\ l$, $\Gamma(x_1) = \textbf{Nat}\ l_1$, $\Gamma(x_2) = \textbf{Nat}\ l_2$, where $l_1 \sqsubseteq \mathtt{A}$ and $l_2 \sqsubseteq \mathtt{A}$. If $l \sqsubseteq \mathtt{A}$, then we have $t_a = \textbf{read}(x_1, v_1)@\textbf{read}(x_2, v_2)@\textbf{writearr}(a, v_1, v_2) = t$, and $i_a = l : y[x_1] := x_2 = i$. For memory, $M^{\star\star} = M'[y \mapsto set(m, v_1, v_2)] \sim_A M[y \mapsto set(m, v_1, v_2)] = M^\star$, where $(m, l) = M'(y) = M(y)$, $(v_1, l_1) = M(x_1)$, and $(v_2, l_2) = M(x_2)$.

If $l = \mathtt{B}$, then $M^{\star\star} = M' \sim_A M \sim_A M[y \mapsto m'] = M^\star$, $i = \epsilon = i_a$, $t = \epsilon = t_a$.

Case l : **if**(x)**then** S_1**else** S_2**.** If $l = \mathtt{B}$, then according to Lemma **??TODO:a new lemma**, $M^{\star}\star = M' \sim_A M \sim_A M^\star$, and $t \equiv \epsilon t_a$, $i \equiv \epsilon i_a$. If $l \sqsubseteq \mathtt{A}$, then $i = l :$ **if**$(x) = i_a$, and $t = \textbf{read}(x, v) = t_a$. Further, $M^{\star\star} = M' \sim_A M = M^\star$. Therefore, the conclusion is also true.

Case l : **while**(x)**do** S'**.** For $S_1 =$ **while**(x)**do** S_b, the proof is very similar to the if-statement. ∎

We are one step away from our main theorem. Lemma 3 says that sim_A can generate correct traces given one declassification. If we have multiple declassification events, then sim_A

$\boxed{\langle M, S, D\rangle \xrightarrow{(i,t)} \langle M', S'\rangle}$

$$\text{Sim-Declass}\ \frac{\begin{array}{c} D = ((x,v),d) \Rightarrow M' = M[x \mapsto v] \\ D = (\epsilon, d) \Rightarrow M' = M \\ \Gamma(x) = \textbf{Nat}\ l \quad t' = select_A(l, \textbf{write}(x,v)) \\ t = y@t' \qquad i = \textbf{declass}(x,y) \end{array}}{\Gamma \vdash \langle M, \mathtt{O} : x := \textbf{declass}_l(y), D\rangle \xrightarrow{(i,t)} \langle M', \mathtt{O} : \textbf{skip}\rangle}$$

$\boxed{\langle M, S, D\rangle \xrightarrow{(i,t)} \langle M', S', D\rangle}$

$$\text{Sim-Concat}\ \frac{\langle M, S, \epsilon\rangle \xrightarrow{(i,t)} \langle M', S', \epsilon\rangle \quad \langle M', S', D\rangle \xrightarrow{(i',t')} \langle M'', S'', D\rangle}{\langle M, S, D\rangle \xrightarrow{(i@i', t@t')} \langle M'', S'', D\rangle}$$

$$\text{Sim-Skip}\ \langle M, l : \textbf{skip}; S, D\rangle \xrightarrow{(\epsilon,\epsilon)} \langle M, S, D\rangle$$

$$\text{Sim-ORAM}\ \frac{\begin{array}{c} \Gamma(x) = \textbf{Oram}\ \mathtt{O} \qquad \Gamma(y) = \textbf{Array}\ l \\ t = select(l, y, y)@x \qquad i = l : \textbf{init}(x,y) \end{array}}{\langle M, \mathtt{O} : x := \textbf{oram}(y), D\rangle \xrightarrow{(i,t)} \langle M, \mathtt{O} : \textbf{skip}, D\rangle}$$

$$\text{Sim-ArrAss}\ \frac{\begin{array}{c} \Gamma, l \vdash \langle M, x_j\rangle \Downarrow_{t_j} v_j\ \ j = 1,2 \quad \Gamma(y) = \textbf{Array}\ l \\ l \sqsubseteq \mathtt{A} \Rightarrow M' = M[y \mapsto set(val(M(y)), v_1, v_2)] \\ \mathtt{B} \sqsubseteq l \Rightarrow M' = M \\ t = t_1@t_2@select_A(l, \textbf{writearr}(y, v_1, v_2), y) \\ i = select_A(l, l : y[x_1] := x_2, l : y[x_1] := x_2) \end{array}}{\langle M, l : y[x_1] := x_2, D\rangle \xrightarrow{(i,t)} \langle M', l : \textbf{skip}, D\rangle}$$

$$\text{Sim-Assign}\ \frac{\begin{array}{c} \Gamma(x) = \textbf{Nat}\ l \quad \Gamma, l \vdash \langle M, e\rangle \Downarrow_{t'} v \\ l \sqsubseteq \mathtt{A} \Rightarrow M' = M[x \mapsto (v, l')] \\ \mathtt{B} \sqsubseteq l \Rightarrow M' = M \\ t = t'@select_A(l, \textbf{write}(x,v), x) \\ i = select_A(l, l : x := e, l : x := e) \end{array}}{\langle M, l : x := e, D\rangle \xrightarrow{(i,t)} \langle M', l : \textbf{skip}, D\rangle}$$

$$\text{Sim-Cond}\ \frac{\begin{array}{c} \Gamma, l \vdash \langle M, x\rangle \Downarrow_{t'} v \\ v = 1 \Rightarrow c = 1 \qquad v \neq 1 \Rightarrow c = 2 \\ i = select_A(l, l : \textbf{if}(x), l : \textbf{if}(x)) \\ t = select_A(l, t', t') \\ l \sqsubseteq \mathtt{A} \Rightarrow S' = S_c \quad l = \mathtt{B} \Rightarrow S' = \mathtt{P} : \textbf{skip} \end{array}}{\langle M, l : \textbf{if}(x)\textbf{then}\ S_1\textbf{else}\ S_2, D\rangle \xrightarrow{(i,t)} \langle M, S', D\rangle}$$

$$\text{Sim-While-True}\ \frac{\begin{array}{c} \Gamma(x) = \textbf{Nat}\ l \qquad l \sqsubseteq \mathtt{A} \\ \Gamma, l \vdash \langle M, x\rangle \Downarrow_t v \quad v \neq 0 \\ S = l : \textbf{while}(x)\textbf{do}\ S' \end{array}}{\Gamma \vdash \langle M, S, D\rangle \xrightarrow{(l:\textbf{while}(x),t)} \langle M, S'; S, D\rangle}$$

$$\text{Sim-While-False}\ \frac{\begin{array}{c} \Gamma(x) = \textbf{Nat}\ l \qquad l \sqsubseteq \mathtt{A} \\ \Gamma, l \vdash \langle M, x\rangle \Downarrow_t v \quad v = 0 \\ S = l : \textbf{while}(x)\textbf{do}\ S' \end{array}}{\Gamma \vdash \langle M, S, D\rangle \xrightarrow{(l:\textbf{while}(x),t)} \langle M, \mathtt{P} : \textbf{skip}, D\rangle}$$

$$\text{Sim-While-Ignore}\ \frac{S = \mathtt{B} : \textbf{while}(x)\textbf{do}\ S'}{\Gamma \vdash \langle M, S, D\rangle \xrightarrow{(\epsilon,\epsilon)} \langle M, l : \textbf{skip}\rangle}$$

$$\text{Sim-Seq}\ \frac{\Gamma \vdash \langle M, S_1, D\rangle \xrightarrow{(i,t)} \langle M', S_1', D\rangle}{\Gamma \vdash \langle M, S_1; S_2, D\rangle \xrightarrow{(i,t)} \langle M', S_1'; S_2, D\rangle}$$

Fig. 16: Operational semantics for statements in sim_A

can simulate them one after another. We know each intermediate memory are Alice-similar to the real memory, which means from Alice's point of view, the memory profile generated by sim_A is consistent with the real memory. Lemma 3 further says that sim_A's program counter and the real program counter are the same after each declassification. Therefore, sim_A can compute $\langle M_{j-1}, S_{j-1}, D_j\rangle \xrightarrow{(i_j,t_j)} \langle M_j, S_j\rangle$ iteratively, where M_0 is the Alice's profile of the initial memory, and S_0 is the program. After n-rounds, i_n and t_n are the instruction traces and memory traces that Theorem 2 requires. Q.E.D.

B. Proof of Theorem 2

Theorem 2 is a nature corollary of the following theorem:

Theorem 4. *If $\Gamma, pc \vdash S$, then either S is l : **skip**, or for any Γ-compatible memory M, there exist $i_a, t_a, i_b, t_b, M', S', D$ such that $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', S'\rangle : D$, M' is Γ-compatible, and $\Gamma, pc \vdash S'$.*

Proof: We prove by induction on the structure of S. If $S = l$: **skip**, then the conclusion is trivial.

If $S = l : x := e$, then $\Gamma(x) = \textbf{Nat}\ l$, $\Gamma \vdash e :$ **Nat** l', $pc \sqcup l' \sqsubseteq l$. We discuss the type of e. If $e = x'$, then we know $\Gamma(x') = \textbf{Nat}\ l'$. Since M is $\Gamma-$compatible, we know $M(x') = (v, l')$, where $v \in$ **Nat**. Therefore, $\langle M, x'\rangle \Downarrow_(t_a, t_b)v$, where $(t_a, t_b) = select(l, \textbf{read}(x', v), x')$, and thus $\langle M, S\rangle \xrightarrow{(i_a,t_a',i_b,t_b')} \langle M', l : \textbf{skip}\rangle : \epsilon$, where $(i_a, i_b) = inst(l, x := e)$, $t_a' = t_a@t_a''$, and $t_b' = t_b@t_b''$, where $(t_a'', t_b'') = select(l, \textbf{write}(x,v), x)$. Further, $M' = M[x \mapsto (v, l)]$. Therefore, M' is also Γ-compatible, and the conclusion holds true. Similarly, we can prove that if $\Gamma \vdash e : \tau$ is derived using T-Const, T-Op, T-Array, or T-Mux, then the conclusion is also true.

If $S = O : x := \textbf{declass}_l(y)$, then $\Gamma(y) = \textbf{Nat}\ O$, $\Gamma(x) =$ **Nat** l where $l \neq \mathbf{O}$, and $pc = \mathbf{P}$. Since M is $\Gamma-$compatible, we know $M(y) = (v, \mathbf{O})$, $M' = M[x \mapsto (v, l)]$. Therefore M'

is $\Gamma-$compatible. Further, $(t'_a, t'_b) = select(l, \textbf{write}(x, v), x)$, $t_a = y@t'_a$, $t_b = y@t'_b$, $i_a = i_b = \mathtt{O} : \textbf{declass}(x, y)$, $D = select(l, (x, v), \epsilon)$, and $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', \mathtt{O} : \textbf{skip}\rangle$, and we know that $\Gamma, \mathtt{P} \vdash \mathtt{O} : \textbf{skip}$. Therefore the conclusion is true.

Similarly, we can prove the conclusion is true for $S = O : x := \textbf{oram}(y)$.

For $S = l : y[x_1] := x_2$, then $\Gamma(y) = \textbf{Array}\ l$, $\Gamma(x_1) = \textbf{Nat}\ l_1$, $\Gamma(x_2) = \textbf{Nat}\ l_2$, and $pc \sqcup l_1 \sqcup l_2 \sqsubseteq l$. Since M is $\Gamma-$compatible, we know $M(y) = (m, l)$, $M(x_1) = (v_1, l_1)$, and $M(x_2) = (v_2, l_2)$. Therefore $M' = M[y \mapsto (set(m, v_1, v_2), l)]$ is also $\Gamma-$compatible. Further, $(t'_a, t'_b) = select(l, \textbf{writearr}(y, v_1, v_2), y)$, $t_a = t_{1a}@t_{2a}@t'_a$, $t_b = t_{1b}@t_{2b}@t'_b$, and $(i_a, i_b) = inst(l, y[x_1] := x_2)$. Therefore, $\langle M, S\rangle \xrightarrow{(i_a,t_a,i_b,t_b)} \langle M', l : \textbf{skip}\rangle$, where we can prove $\Gamma, pc \vdash l : \textbf{skip}$ easily. Therefore, the conclusion is true.

For $S = l : \textbf{if}(x)\textbf{then}\ S_1\ \textbf{else}\ S_2$, we have $\Gamma(x) = \textbf{Nat}\ l$. Therefore $M(x) = (v, l)$. If $v = 1$, then $\langle M, S\rangle \xrightarrow{i_a,t_a,i_b,t_b} \langle M, S_1$ where $(i_a, i_b) = inst(l, \textbf{if}(x))$ and $(t_a, t_b) = select(l, \textbf{read}(x, v), x)$. Further, we know $\Gamma, l \vdash S_1$. Since $pc \sqsubseteq l$, it is easy to prove by induction that $\Gamma, pc \vdash S_1$ is true as well. Therefore, the conclusion is true. On the other hand, if $v \neq 1$, then $\langle M, S\rangle \xrightarrow{i_a,t_a,i_b,t_b} \langle M, S_2$. We can also prove the conclusion.

The proof for $S = l : \textbf{while}(x)\textbf{do}\ S'$ is similar to the branching-statement by using rule S-While-True and S-While-False.

For $S = S_1; S_2$, then we know $\Gamma \vdash S_1$. The conclusion directly follows the induction assumption by applying rule S-Seq and rule S-Skip. ■

C. Proof of Theorem 3

We shall show that when P is Γ-simulatable, we can provide a hybrid world protocol Π that execute the ideal world functionality $\mathcal{F}_0$. The protocol follows the semantics defined in Figure 17, Figure 18 and Figure 19 (for hybrid world). Π will call the following ideal world functionalities $\mathcal{F}_{op}$, $\mathcal{F}_{\text{oram}}$, $\mathcal{F}_{\text{oram}}$, $\mathcal{F}_{\text{mux}}$, and $\mathcal{F}_{\text{declass}}$. The input, output, and meaning of these functionalities are explained in Table III. In the table, there are parameterized functionalities, where the type parameter T has the following meaning: if $T = \mathtt{A}$ (or $\mathtt{B}$) it means Alice (or Bob) provides the data; if $T = \mathtt{P}$, then the data is public; if $T = \mathtt{O}$, then the data is secret-shared between Alice and Bob. For example, the functionality $\mathcal{F}^{(T_1,T_2)}_{\text{addition}}$, where $T_1 = \mathtt{A}$ and $T_2 = \mathtt{B}$, is the functionality that Alice provides v_1 and Bob provides v_2, while the result $v_1 + v_2$ is shared between Alice and Bob. Another example is that for $\mathcal{F}^{(\mathtt{O},\mathtt{O})}_{\text{multiplication}}$ means that the two inputs v_1 and v_2 are secret-shared between Alice and Bob, and the result $v_1 \times v_2$ is shared between Alice and Bob. For the output of all functionalities except $\mathcal{F}^{T_O}_{\text{declass}}$ is secret-shared between Alice and Bob, but $\mathcal{F}^{T_O}_{\text{declass}}$'s input is secret shared between Alice and Bob, and it outputs to the party T_O. All these parameters are determined at compile time.

Local rules contain three kinds of rules for Alice's local mode, Bob's local mode, and public mode respectively. Rules for Alice are in the form of

$$\langle M_P, M_A, i, t\rangle \to \langle M_P, M'_A\rangle$$

which means Alice keeps track of public memory M_P, and M_A, and Alice execute (i, t), then it will result in $\langle M_P, M'_A\rangle$, which means that M_P and S_A will not be changed. Specifically, there are four rules. Rule A-Assign deals with $A : x := e$, then we know t must be $eval^{\Gamma}_A(e)@\textbf{write}(x, v)$. Here, $eval^{\Gamma}_A(e)$ is used to indicate the corresponding memory trace of e observed by Alice under Γ. Particularly, for $e = x$, $eval^{\Gamma}_A(e)$ is determined by the following function:

$$eval^{\Gamma}_A(x) = \begin{cases} read(x, v) & \Gamma(x) \sqsubseteq \mathtt{A} \\ \epsilon & \Gamma(x) = \mathtt{B} \\ x & \Gamma(x) = \mathtt{O} \end{cases}$$

Further, we have

$$eval^{\Gamma}_A(x\ op\ y) = eval^{\Gamma}_A(x)@eval^{\Gamma}_A(y)$$

$$eval^{\Gamma}_A(\textbf{mux}(a, b, c)) = eval^{\Gamma}_A(a)@eval^{\Gamma}_A(b)@eval^{\Gamma}_A(c)$$

Finally, for array access, we have

$$eval^{\Gamma}_A(a[x]) = \begin{cases} eval^{\Gamma}_A(x)@\textbf{read}(a, v_1, v_2) & \Gamma(a) \sqsubseteq \mathtt{A} \\ eval^{\Gamma}_A(x) & \Gamma(a) = \mathtt{B} \\ eval^{\Gamma}_A(x)@a & \Gamma(a) = \mathtt{O} \end{cases}$$

In Alice's local rule, Alice need not do computation over $eval^{\Gamma}_A(e)$, since $\textbf{write}(x, v)$ contains all information that Alice need to execute the rule. This rule is to guarantee that the correct memory trace is processed, i.e. in front of $\textbf{write}(x, v)$, the memory trace must be $eval^{\Gamma}_A(e)$. So Alice simply modifies x's value in her local memory to v.

Rule A-ArrAss deals with array assignment, i.e. $i = \mathtt{A} : a[x_1] := x_2$. In this case, we know $t = eval^{\Gamma}_A(x_1)@eval^{\Gamma}_A(x_2)@\textbf{write}(a, v_1, v_2)$. (Actually, we can reason that $eval^{\Gamma}_A(x_1) = \textbf{read}(x_1, v_1)$ and $eval^{\Gamma}_A(x_2) = \textbf{read}(x_2, v_2)$.), then Alice will change the array a's v_1-th value to v_2. Rule A-Cond and rule A-While do not change M_A, but both of them will require that the trace $\textbf{read}(x, v)$ is processed by Alice. Bob's and public local rules are similar, and functions $eval^{\Gamma}_B$ and $eval^{\Gamma}_P$ are defined similar to $eval^{\Gamma}_A$.

For hybrid world protocol is composed by a set of hybrid world rules to derive statements in the format of

$$\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i^e_a,t^e_a,i^e_b,t^e_b)} \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i'_b, t'_b\rangle, M'_P, M'_S : D$$

Such statement means that Alice keeps trace of M_A, and Bob keeps trace of M_B; M_P is public to both parties, and M_S is secret-shared between Alice and Bob (each array in M_S is maintained by an ORAM functionality); By executing one step, the memory is changed to M'_P, M'_A, M'_B, and S'_A and S'_B, while exposing traces (i^e_a, t^e_a) to Alice and (i^e_b, t^e_b) to Bob.

The protocol Π runs as follows

1) Given program P, input memory M, and a list of declassification $D^0, ..., D^{n-1}$, both Alice and Bob will run their simulators to get $(i^j_a, t^j_a) = sim_A(M[\mathtt{P}], M[\mathtt{A}], D^0_A, ..., D^j_A)$ (for $j = 0, ..., n-1$),

Functionalities	Input	Output	Comments
$\mathcal{F}_{op}^{(T_1,T_2)}$	v_1, v_2	$v = v_1\ op\ v_2$	For each binary operation op, there are 16 ideal world functionalities, parameterized by T_1 and T_2.
$\mathcal{F}_{\text{mux}}^{(T_1,T_2,T3)}$	v_1, v_2, v_3	$v = \textbf{mux}(v_1, v_2, v_3)$	Standard multiplex operator, compute Similar to $\mathcal{F}_{op}$, there are 64 functionalities parameterized by T_1, T_2 and T_3.
$\mathcal{F}_{\text{oram}}^{m}$	$\texttt{read}^{T}, idx$	$v = get(m, idx)$	The ORAM functionality (for each array m) maintains the state of an ORAM m. If both parties send a read instruction, and party T sends the index idx, then $get(m, idx)$ is returned. The state is not changed.
	$\texttt{write}^{T_1,T_2}, idx, v$	$m' = set(m, idx, v)$	If both parties send a write instruction, party T_1 sends the index idx and party T_2 sends the value v, then the state changed to $set(m, idx, v)$ is returned
	$\texttt{init}^{T}, m'$	m'	Party T sends an initialization request to initialize ORAM m using m'.
$\mathcal{F}_{\text{declass}}^{T_O}$	v_i	$v_o = v_i$	Declassify secret-shared value v_i to v_o of party T

TABLE III: Ideal World Functionalities

and $(i_b^j, t_b^j) = sim_B(M[\mathtt{P}], M[\mathtt{B}], D_B^0, ..., D_B^0)$ (for $j = 0, ..., n-1$)

2) set $i_a = i_a^0@...i_a^{n-1}$, $t_a = t_a^0@...t_a^{n-1}$, $i_b = i_b^0@...i_b^{n-1}$, and $t_b = t_b^0@...t_b^{n-1}$.

3) Alice and Bob will run

$$\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i_a^e, t_a^e, i_b^e, t_b^e)} \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i'_b, t'_b\rangle, M_P, M_S : D$$

based on the rules that will be expained in detail below.

We first explain how the protocol proceeds based on rules, and then show the correctness and security of this protocol.

Rule Local-A says that if Alice can execute $\langle M_A, i_a, t_a\rangle$ to M'_A locally, then she will do so. Local-B is the same. For rule Local-P is similar but it requires both Alice and Bob execute over the same instruction trace and memory trace. Rule Seq-A says that if Alice has a instruction trace $i_1@i_2$ and memory trace $t_1@t_2$, then she will interact with Bob as executing (i_1, t_1) first, and (i_2, t_2) afterwards. So that Alice need not be aware of Bob's status to proceed the protocol. Seq-B is the same for Bob. Rule Concat is used to concat two steps of the protocol into a big step.

Figure 19 talks about how a single secure computation is performed between Alice and Bob. Rules starting with Assign, talks about how $\mathtt{0} : x := e$ is executed for each type of e. For a single variable $e = y$, y's value will be secret-shared among Alice and Bob. If $e = a\ op\ b$, then one of the binary functionalities $\mathcal{F}_{op}$ will be called depending on who has variable a and b. Mux is similar to Op except it has three parameters. Finally, for Array, if $\Gamma(a) \neq \mathtt{0}$, then the party holding a will secret share the value among the two parties; otherwise, an ORAM read will be performed to get the result.

Rule AssAss talks about how the array assign instruction is executed, which is very similar to Assign, but employing the ORAM write functionality. For Cond and While, nothing happends except requiring the memory traces to be x. Rule ORAM talks about how ORAM is initialized, while rule Declass deals with the declassification instruction.

Correctness

The correctness of this protocol can be shown by the following two lemmas:

Lemma 4. *Given an environment Γ, a program P, and an memory $M = M_P \cup M_A \cup M_B \cup M_S$, and f is a arbitrary list of states for each ORAM. If $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D^j$, $D^j \neq \epsilon$ for $j = 0, ..., n-1$, where $M_0 = M$ and $P_0 = P$, and $\langle M_n, P_n\rangle \xrightarrow{(i_a^n, t_a^n, i_b^n, t_b^n)} \langle M', P'\rangle : D$, where $D \neq \epsilon$ or $P' =$ **skip** then run the protocol Π, then either $i_a = i_b = \epsilon$ and $t_a = t_b = \epsilon$, or there is at least one way to run the protocol so that $\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M_P, M_S : D$, where $i_a = i_a^0@...@i_a^n$, $t_a = t_a^0@...@t_a^n$, $i_b = i_b^0@...@i_b^n$, $t_b = t_b^0@...@t_b^{n-1}$, such that (1) (declassification equivalence) $D \equiv D^0@...@D^{n-1}$; and (2) (memory equality) $M' = M'_P \cup M'_A \cup M'_B \cup S'_A \oplus S'_B$.*

Proof: Proof can be done by induction on the structure of P. If P is $S_1; S_2$, then we know the following condition happens: There is $k \in \{0, .., n+1\}$, such that $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D_j$, for $j = 0, ..., k-1$, where $P_0 = S$, $\langle M_k, P_k\rangle \xrightarrow{(i_a^p, t_a^p, i_b^p, t_b^p)} \langle M'_k, \textbf{skip}\rangle : \epsilon$, $\langle M'_k, S_2\rangle \xrightarrow{(i_a^q, t_a^q, i_b^q, t_b^q)} \langle M_{k+1}, P_{k+1}\rangle : D_k$, and $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D_j$, for $j = k+1, ..., n-1$, and $i_a^k = i_a^p@i_a^q$, $t_a^k = t_a^p@t_a^q$, $i_b^k = i_b^p@i_b^q$, and $t_b^k = t_b^p@t_b^q$. Notice that k can be $n+1$, in which case S_1 is never executed to **skip**, and S_2 is never executed.

In this case, we know $i_a = i_a^{(1)}@i_a^{(2)}$, where $i_a^{(1)} = i_a^0@...@i_a^{k-1}@i_a^p$, and $i_a^{(2)} = i_a^q@i_a^{k+1}@...@i_a^n$, and similarly, $t_a = t_a^{(1)}@t_a^{(2)}$, $i_b = i_b^{(1)}@i_b^{(2)}$, and $t_b = t_b^{(1)}@t_b^{(2)}$ defined in a similar way. By induction, we have

$$\langle M_A, i_a^{(1)}, t_a^{(1)}\rangle, \langle M_B, i_b^{(1)}, t_b^{(1)}\rangle, M_P, M_S \xrightarrow{(i'^1_a, t'^1_a, i'^1_b, t'^1_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M'_P, M'_S : D^1$$

where $D^1 = D_0@...@D_{k-1}$, and

$$\langle M'_A, i_a^{(2)}, t_a^{(2)}\rangle, \langle M'_B, i_b^{(2)}, t_b^{(2)}\rangle, M'_P, M'_S \xrightarrow{(i'^2_a, t'^2_a, i'^2_b, t'^2_b)} \langle M''_A, \epsilon, \epsilon\rangle, \langle M''_B, \epsilon, \epsilon\rangle, M''_P, M''_S : D^2$$

Alice's local execution

$$\text{A-Assign}\ \frac{i = \texttt{A} : x := e \quad t = eval_A^{\Gamma}(e)@\textbf{write}(x, v) \quad M'_A = M_A[v/x]}{\langle M_A, i, t\rangle \rightarrow \langle M'_A\rangle}$$

$$\text{A-ArrAss}\ \frac{\begin{array}{ccc} i = \texttt{A} : a[x_1] := x_2 & t = \textbf{read}(x_1, v_1)@\textbf{read}(x_2, v_2)@\textbf{write}(a, v_1, v_2) & \\ m = M_A(a) & m' = set(m, v_1, v_2) & M'_A = M_A[m'/a] \end{array}}{\langle M_A, i, t\rangle \rightarrow \langle M'_A\rangle}$$

$$\text{A-Cond}\ \frac{i = \texttt{A} : \textbf{if}(x) \quad t = \textbf{read}(x, v)}{\langle M_A, i, t\rangle \rightarrow \langle M_A\rangle}$$

$$\text{A-While}\ \frac{i = \texttt{A} : \textbf{while}(x) \quad t = \textbf{read}(x, v)}{\langle M_A, i, t\rangle \rightarrow \langle M_A\rangle}$$

Bob's local execution

$$\text{B-Assign}\ \frac{i = \texttt{B} : x := e \quad t = eval_B^{\Gamma}(e)@\textbf{write}(x, v) \quad M'_B = M_B[v/x]}{\langle M_B, i, t\rangle \rightarrow \langle M'_B\rangle}$$

$$\text{B-ArrAss}\ \frac{\begin{array}{ccc} i = \texttt{B} : a[x_1] := x_2 & t = \textbf{read}(x_1, v_1)@\textbf{read}(x_2, v_2)@\textbf{write}(a, v_1, v_2) & \\ m = M_B(a) & m' = set(m, v_1, v_2) & M'_B = M_A[m'/a] \end{array}}{\langle M_B, i, t\rangle \rightarrow \langle M'_B\rangle}$$

$$\text{B-Cond}\ \frac{i = \texttt{B} : \textbf{if}(x) \quad t = \textbf{read}(x, v)}{\langle M_B, i, t\rangle \rightarrow \langle M_B\rangle}$$

$$\text{B-While}\ \frac{i = \texttt{B} : \textbf{while}(x) \quad t = \textbf{read}(x, v)}{\langle M_B, i, t\rangle \rightarrow \langle M_B\rangle}$$

Public execution

$$\text{P-Assign}\ \frac{i_a = i_b = \texttt{P} : x := e \quad t_a = t_b = eval_P^{\Gamma}(e)@\textbf{write}(x, v) \quad M'_P = M_P[v/x]}{\langle M_P, i_a, t_a, i_b, t_b\rangle \rightarrow \langle M'_P\rangle}$$

$$\text{P-ArrAss}\ \frac{\begin{array}{ccc} i = \texttt{P} : a[x_1] := x_2 & t = \textbf{read}(x_1, v_1)@\textbf{read}(x_2, v_2)@\textbf{write}(a, v_1, v_2) & \\ m = M_P(a) & m' = set(m, v_1, v_1) & M'_P = M_A[m'/a] \end{array}}{\langle M_P, i, t\rangle \rightarrow \langle M'_P\rangle}$$

$$\text{P-Cond}\ \frac{i_a = i_b = \texttt{P} : \textbf{if}(x) \quad t_a = t_b = \textbf{read}(x, v)}{\langle M_P, i_a, t_a, i_b, t_b\rangle \rightarrow \langle M_P\rangle}$$

$$\text{P-While}\ \frac{i_a = i_b = \texttt{P} : \textbf{while}(x) \quad t_a = t_b = \textbf{read}(x, v)}{\langle M_P, i_a, t_a, i_b, t_b\rangle \rightarrow \langle M_P\rangle}$$

Fig. 17: Local execution

where $D^2 = D_k@...@D_n$. Then by applying rule Seq-A, Seq-B, and Concat, we have

$$\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i'^1_a@i'^2_a, t'^1_a@t'^2_a, i'^1_b@i'^2_b, t'^1_b@t'^2_b)} \langle M''_A, \epsilon, \epsilon\rangle, \langle M''_B, \epsilon, \epsilon\rangle, M''_P, M''_S : D^1@D^2$$

where $D = D^1@D^2$. Further, by induction, we know $M'_k = M'_P \cup M'_A \cup M'_B \cup M'_S$, and further $M' = M''_P \cup M''_A \cup M''_B \cup M''_S$.

If $P = l$: **skip**, then $i_a = i_b = \epsilon$, and $t_a = t_b = \epsilon$. The conclusion is trivial.

If $P = l : x := e$, then $i_a = i_b = l : x := e$. If $l = \texttt{A}$, then $\Gamma(x) = \texttt{A}$, $t_a = eval_A^{\Gamma}(e)@\textbf{write}(x, v)$, and $t_b = \epsilon$. Then by applying rule A-Assign, Local-A, we can get the conclusion. Similarly, if $l = \texttt{B}$ or $l = \texttt{P}$, the conclusion is also true. If $l = \texttt{O}$, then we can apply one of rules Assign-Var, Assign-Array, Assign-Op, and Assign-Mux to get the conclusion.

If $P = l : a[x] := y$, similar to the above condition, if $l = \texttt{A}$, then we can apply rule A-Array, and Local-A to get the conclusion. The discussion for $l = \texttt{B}$ and $l = \texttt{P}$ is similar, and for $l = \texttt{O}$, we can apply rule ArrAss to get the conclusion.

If $P = l$: **if** (x) **then** S_1 **else** S_2, then we discuss based on

Protocol

$$\text{Local-A}\ \frac{\langle M_A, i_a, t_a\rangle \to \langle M'_A\rangle}{\langle M_A, i_a, t_a\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M_S \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S : \epsilon}$$

$$\text{Local-B}\ \frac{\langle M_B, i_b, t_b\rangle \to \langle M'_B\rangle}{\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(\epsilon, \epsilon, i_b, t_b)} \langle M_A, i_a, t_a\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M_P, M_S : \epsilon}$$

$$\text{Local-P}\ \frac{\langle M_P, i_a, t_a, i_b, t_b\rangle \to \langle M'_P\rangle}{\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i_a, t_a, i_b, t_b)} \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M'_P, M_S : \epsilon}$$

$$\text{Seq-A}\ \frac{\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i^1_a, t^1_a, i^1_b, t^1_b)} \langle M'_A, i', t'\rangle, \langle M'_B, i'_b, t'_b\rangle, M'_P, M'_S : D}{\langle M_A, i_a @ i'_a, t_a @ t'_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i^1_a, t^1_a, i^1_b, t^1_b)} \langle M'_A, i' @ i'_a, t' @ t'_a\rangle, \langle M'_B, i'_b, t'_b\rangle, M'_P, M'_S : D}$$

$$\text{Seq-B}\ \frac{\langle M_A, i_a, t_a\rangle, \langle M_B, S_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i^1_a, t^1_a, i^1_b, t^1_b)} \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i', t'\rangle, M'_P, M'_S : D}{\langle M_A, i_a, t_a\rangle, \langle M_B, S_B, i_b @ i'_b, t_b @ t'_b\rangle, M_P, M_S \xrightarrow{(i^1_a, t^1_a, i^1_b, t^1_b)} \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i' @ i'_b, t' @ t'_b\rangle, M'_P, M'_S : D}$$

$$\text{Concat}\ \frac{\begin{array}{c}\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i^1_a, t^1_a, i^1_b, t^1_b)} \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i'_b, t'_b\rangle, M'_P, M'_S : D_1 \\ \langle M'_A, i'_a, t'_a\rangle, \langle M'_B, i'_b, t'_b\rangle, M'_P, M'_S \xrightarrow{(i^2_a, t^2_a, i^2_b, t^2_b)} \langle M''_A, i''_a, t''_a\rangle, \langle M''_B, i''_b, t''_b\rangle, M''_P, M''_S : D_2\end{array}}{\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, f \xrightarrow{(i^1_a @ i^2_a, t^1_a @ t^2_a, i^1_b @ i^2_b, t^1_b @ t^2_b)} \langle M''_A, i''_a, t''_a\rangle, \langle M''_B, i''_b, t''_b\rangle, M''_P, M''_S : D_1 @ D_2}$$

Fig. 18: Protocol Π (Part I)

l. If $l = \texttt{A}$, then we know $i_a = \textbf{if}(x)@i^\star_a$, $t_a = eval^\Gamma_A(x)@t^\star_a$, $i_b = i^\star_b$, and $t_b = t^\star_b$, and thus by induction assumption, we know

$$\langle M_A, i^\star_a, t^\star_a\rangle, \langle M_B, i^\star_b, t^\star_b\rangle, M_P, M'_S \xrightarrow{(i'^\star_a, t'^\star_a, i'^\star_b, t'^\star_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M_P, M'_S : D$$

Therefore, we know

$$\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M'_S \xrightarrow{(\textbf{if}(x)@i'^\star_a, eval^\Gamma_A(x)@t'^\star_a, i'^\star_b, t'^\star_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M'_P, M'_S : D$$

Similarly, we can prove for $l \in \{\mathbf{B}, \mathbf{O}, \mathbf{P}\}$.

The discussion for $P = l : \textbf{while}\ (x)\ \textbf{do}\ S$ is very similar for the if-statement.

If $P = l : x := \textbf{declass}(y)$, then $i_a = i_b = \texttt{O} : \textbf{decalss}(x, y)$, $(t'_a, t'_b) = select(l, \textbf{write}(x, v), x)$, $t_a = y@t'_a$, and $t_b = y@t'_b$. Further, if $\Gamma(x) = \texttt{A}$, then $D = ((x, v), \epsilon)$; if $\Gamma(x) = \texttt{B}$, then $D = (\epsilon, (x, v))$; if $\Gamma(x) = \texttt{O}$, then $D = ((x, v), (x, v))$. In any case, we can apply rule Declass to get our conclusion.

Similarly, for $P = l : x := \textbf{oram}(y)$, we can apply rule ORAM to get the conclusion. ■

Lemma 5. *Given an environment Γ, a program P, and an memory $M = M_P \cup M_A \cup M_B \cup M_S$. If $\langle M_j, P_j\rangle \xrightarrow{(i^j_a, t^j_a, i^j_b, t^j_b)} \langle M_{j+1}, P_{j+1}\rangle : D^j$, $D^j \neq \epsilon$ for $j = 0, ..., n-1$, where $M_0 = M$ and $P_0 = P$, and $\langle M_n, P_n\rangle \xrightarrow{(i^n_a, t^n_a, i^n_b, t^n_b)} \langle M', P'\rangle : D$, where $D \neq \epsilon$ or $P =$ **skip**, then run the protocol $\mathcal{F}$, if we get $\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M'_P, M'_S : D$, where $i_a = i^0_a @ ... @ i^n_a$, $t_a = t^0_a @ ... @ t^n_a$, $i_b = i^0_b @ ... @ i^n_b$, $t_b = t^0_b @ ... @ t^{n-1}_b$, then (trace equivalence) $i_a \equiv i'_a$, $t_a \equiv t'_a$, $i_b \equiv i'_b$, and $t_b \equiv t'_b$.*

Proof: (sketch) Proof can be trivially done by induction of how the derivation is made. ■

Security

We first need to prove that $\text{EXEC}^{\mathcal{F}}_{\pi}$ (here, $\mathcal{F}$ denotes the set of all functionalities mentionedin Table III) securely emulates $\text{IDEAL}^{\mathcal{F}_0}$. In our setting, for Alice, Bob is a semi-honest adversary, who can manipulate his memory. Therefore, all we need to prove is that for Alice, when executing Π over $\langle M_A, i_a, t_a\rangle, \langle M_B, i_b, t_b\rangle, M_P, M_S$, the adversary Bob will learn $i'_b \equiv i_b$ and $t'_b \equiv t_b$, and a list of declassifications $D^0_B @ ... @ D^n_B$. In the ideal world, $\text{IDEAL}^{\mathcal{F}_0}$, we can simply build an adversary Bob to learn the output by executing $\langle M_P \cup M_A \cup M_B \cup M_S, P$, which is i_b, t_b and $D'_B = D^0_B @ ... @ D^n_B$. They are distinguishable to i_b, t_b and D_B.

Then, we can rely on Canetti's sequential composibility framework to argue the security of the entire protocol.

Protocol

$$\text{Assign-Var}\ \frac{\begin{array}{c} i = \mathtt{0} : x := y \qquad t_a = eval^{\Gamma}_A(y)@x \qquad t_b = eval^{\Gamma}_B(y)@x \\ M = M_P \cup M_A \cup M_B \cup M_S \qquad v_1 = M(y) \qquad M'_S = M_S[v/x] \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \to \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{Assign-Op}\ \frac{\begin{array}{c} i = \mathtt{0} : x := y\ op\ z \qquad t_a = eval^{\Gamma}_A(y)@eval^{\Gamma}_A(z)@x \qquad t_b = eval^{\Gamma}_B(y)@eval^{\Gamma}_B(z)@x \\ M = M_P \cup M_A \cup M_B \cup M_S \qquad v = \mathcal{F}_{op}^{(\Gamma(y),\Gamma(z))}(M(x), M(y)) \qquad M'_S = M_S[v/x] \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_S \to \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M'_P, M'_S : \epsilon}$$

$$\text{Assign-Array}\ \frac{\begin{array}{c} i = \mathtt{0} : x := a[y] \qquad t_a = eval^{\Gamma}_A(a[y])@x \qquad t_b = eval^{\Gamma}_B(a[y])@x \\ M = M_P \cup M_A \cup M_B \cup M_S \qquad M'_S = M_S[v/x] \\ \Gamma(a) = \mathtt{A} \wedge \Gamma(a) = \mathtt{P} \Rightarrow t_a = \textbf{write}(a, v_y, v) \\ \Gamma(a) = \mathtt{B} \Rightarrow t_b = \textbf{write}(a, v_y, v) \\ \Gamma(a) = \mathtt{0} \Rightarrow m = M(a) \wedge v = \mathcal{F}_{\text{oram}}^{M_S(a)}(\mathtt{read}^{\Gamma(y)}, y) \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle \to \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{Assign-Mux}\ \frac{\begin{array}{c} i = \mathtt{0} : x := \textbf{mux}(a, b, c) \qquad t_a = eval^{\Gamma}_A(a)@eval^{\Gamma}_A(b)@eval^{\Gamma}_A(c)@x \\ t_b = eval^{\Gamma}_B(a)@eval^{\Gamma}_B(b)@eval^{\Gamma}_B(c)@x \qquad M = M_P \cup M_A \cup M_B \cup M_S \\ v = \mathcal{F}_{\text{mux}}^{(\Gamma(a),\Gamma(b),\Gamma(c))}(M(a), M(b), M(c)) \qquad M'_S = M_S[v/x] \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle \to \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{ArrAss}\ \frac{\begin{array}{c} i = \mathtt{0} : a[x_1] := x_2 \qquad t_a = eval^{\Gamma}_A(x_1)@eval^{\Gamma}_A(x_2)@a \qquad t_b = eval^{\Gamma}_B(x_1)@eval^{\Gamma}_B(x_2)@a \\ M = M_P \cup M_A \cup M_B \cup M_S \qquad v_1 = M(x_1) \qquad v_2 = M(x_2) \\ M'_S = \mathcal{F}_{\text{oram}}^{M_S(a)}(\mathtt{write}^{(\Gamma(x_1),\Gamma(x_2))}, v_1, v_2) \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \xrightarrow{(i,t_a,i,t_b)} \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{Cond}\ \frac{i = \mathtt{0} : \textbf{if}(x) \qquad t_a = t_b = x}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \xrightarrow{(i,t_a,i,t_b)} \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{While}\ \frac{i = \mathtt{0} : \textbf{while}(x) \qquad t_a = t_b = x}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \xrightarrow{(i,t_a,i,t_b)} \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

$$\text{Declass}\ \frac{\begin{array}{c} i = \mathtt{0} : \textbf{declass}(x, y) \qquad t_a = y@eval^{\Gamma}_A(x) \qquad t_b = y@eval_B(x) \\ v = \mathcal{F}_{\text{declass}}^{\Gamma(x)}(M_S(y)) \qquad \Gamma(x) = \mathtt{A} \Rightarrow D = ((x, v), \epsilon) \wedge M'_P = M_P \wedge M'_A = M_A[v/x] \wedge M'_B = M_B \\ \Gamma(x) = \mathtt{B} \Rightarrow D = (\epsilon, (x, v)) \wedge M'_P = M_P \wedge M'_A = M_A \wedge M'_B = M_B[v/x] \\ \Gamma(x) = \mathtt{P} \Rightarrow D = ((x, v), (x, v)) \wedge M'_P = M_P[v/x] \wedge M'_A = M_A \wedge M'_B = M_B \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \xrightarrow{(i,t_a,i,t_b)} \langle M'_A, \epsilon, \epsilon\rangle, \langle M'_B, \epsilon, \epsilon\rangle, M'_P, M_S : D}$$

$$\text{ORAM}\ \frac{\begin{array}{c} i = \mathtt{0} : \textbf{init}(x, y) \qquad t_a = eval^{\Gamma}_A(y)@x \qquad t_b = eval^{\Gamma}_B(y)@x \\ M = M_P \cup M_A \cup M_B \cup M_S \qquad m = \mathcal{F}_{\text{oram}}^{x}(\mathtt{init}^{\Gamma(y)}, M(y)) \end{array}}{\langle M_A, i, t_a\rangle, \langle M_B, i, t_b\rangle, M_P, M_S \xrightarrow{(i,t_a,i,t_b)} \langle M_A, \epsilon, \epsilon\rangle, \langle M_B, \epsilon, \epsilon\rangle, M_P, M'_S : \epsilon}$$

Fig. 19: Protocol Π (Part II)