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 compared to both circuit-model secure computation and the state-of-art 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 [31], research on secure two-party computation—especially in the semi-honest model we consider here—has flourished, resulting in ever more efficient protocols [4], [10], [15], [32] as well as several practical implementations [6], [11]–[13], [16], [21]. Since the first system for general-purpose secure two-party computation was built in 2004 [21], efficiency has improved substantially [4], [13].

Almost all previous implementations of general-purpose secure computation assume the underlying computation is represented as a *circuit*. While theoretical developments using circuits are sensible (and common), compiling typical programs, which assume a von Neumann-style Random Access Machine (RAM) model, to efficient circuits can be challenging. One significant challenge is handling *dynamic memory accesses* to an array in which the memory location being read/written depends on secret inputs. A typical program-to-circuit compiler typically makes an entire copy of the array upon every dynamic memory access, thus resulting in a huge circuit when the data size is large. Theoretically speaking, generic approaches for translating RAM programs into circuits incur, in general, $O(TN)$ blowup in efficiency, where T is an upper bound on the program's running time, and N is the memory size.

To address the above limitations, researchers have more recently considered secure computation that works directly in the RAM model [10], [20]. 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 of providing 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.[1] In particular, 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, and more. 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, the standard implementation of RAM-SC entails placing all data and instructions inside a single Oblivious RAM. The secure evaluation of one instruction then requires *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

[1] 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.

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 capable of expressing RAM-SC tasks more efficiently by avoiding expensive next-instruction circuits and minimizing ORAM operations when there is no risk to 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., it is part of a linear scan of the array), then the array need not be placed into ORAM.

We formally define the security property 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. SCVM 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 explored (in the circuit model) by Kerschbaum [15] and Rastogi et al. [25] 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 RAM-model secure 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 notation adjusted for our purposes. We compare our scheme against the one presented here in Section VI-C.

A key underlying building block of RAM-model secure computation is *Oblivious RAM (ORAM)* which was initially proposed by Goldreich and Ostrovsky [8] and later improved in a sequence of works [9], [17], [27], [28], [30]. 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.

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* (like a 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

Existing RAM-model secure computation proceeds as in Figure 1. The entire memory denoted `mem`, containing both program instructions and data, is placed in ORAM, and the ORAM is secret-shared between the two participating parties as discussed above, e.g., using a simple XOR-based secret-sharing scheme. With ORAM, a memory access thus requires each party to access the elements of their respective arrays at pseudorandom locations (the addresses are dictated by the ORAM algorithm), and the value stored at each position is then obtained by XORing the values read by each of the parties. Alternatively, the server can hold an encryption of the ORAM array, and the client holds the key. The latter was done by Gordon et al. to ensure that one party holds only $O(1)$ state. All CPU states, including `pc`, `reg`, `I`, `rdata`, `wdata`, `raddr`, and `waddr`, are also secret-shared between the two parties.

In Figure 1, each step of the computation must be done using some secure computation subprotocol. In particular, SC-U is a secure computation protocol that securely evaluates the universal next instruction circuit, and SC-ORAM is a secure computation protocol that securely evaluates the ORAM

/* All variables, including `mem`, `pc`, `I`, `rdata`, `reg`, `wdata`, `raddr`, and `waddr` are secret-shared between the two parties. */

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

instr. fetch phase:	`I` $\leftarrow$ `ORAM.Read(mem, pc)`	*//Protocol SC-ORAM*
CPU phase:	(`pc`, `raddr`, `waddr`, `wdata`, `reg`) $\leftarrow U$(`I, reg, rdata`)	*//Protocol SC-U*
data read phase:	`rdata` $\leftarrow$ `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. All data are secret-shared.

	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

algorithm. For `ORAM.Read`, each party supplies a secret share of the `raddr`, and during the course of the protocol, the `ORAM.Read` protocol will emit obfuscated addresses for each party to read from. At the end of the protocol, each party obtains a share of the fetched data. For `ORAM.Write`, each party supplies a secret share of `waddr` and `wdata`, and during the course of the protocol, the `ORAM.Read` protocol will emit obfuscated addresses for each party to write to, and secret shares of values to write to those addresses.

Scenarios for RAM-model secure computation. While Gordon et al. describe RAM-model secure computation mainly for 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. [22] proposed a way to reason about the amount of information declassified by the result of a secure computation, and Rastogi et al. [25] 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 (though their results would not necessarily be accepted by our type system, whose improved precision would be future work). Concurrently with our work, Rastogi et al. [24] developed Wysteria, a programming language for *mixed mode* secure multiparty computations, which consist of local computations intermixed with joint, secure ones. While this high-level idea is similar to our work, the details are very different. For example, they do not provide a simulatability theorem (they propose to accept results from the analysis of Rastogi et al. [25]) and are focused more at usability.

Zahur and Evans [32] 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 [1].

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

This section describes our approach to optimize RAM-model secure computation. Our key idea is use static analysis during compilation 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 reveal 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 to the following:

```
t1 := s;
t2 := a+b;
t3 := a-b;
t4 := mux(t1, t2, t3);
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 *program-counter security* proposed by Molnar et al. [23] (for a different application).

B. Memory-Trace Obliviousness

Using ORAM for memory accesses is also a heavyweight operation in RAM-model secure computation. The standard 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. [18] recently observe that not all access patterns of a program are sensitive. 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. The figure shows the code that computes shortest paths (Bob's inputs are elided).

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, and presents our formal results.

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 $\mathcal{F}$

```
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 && t6;
        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:=!t7;
        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 `n`, `i`, `j` and others boxed in white background are public variables. All other variables are secret-shared between the two parties.

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 formally defines our security property, Γ-*simulatability*. Informally, a program is secure if each party, starting with its own inputs, 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 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.

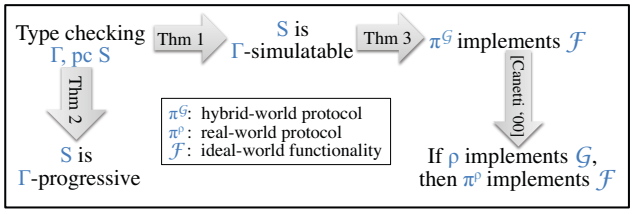


Fig. 3: Formal results.

A. Syntax

The syntax of SCVM is given in Figure 4. 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 $\mathtt{P} \sqsubseteq l \sqsubseteq \mathtt{O}$ for $l \in \{\mathtt{A}, \mathtt{B}\}$. 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 will be added in Section V.

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	::=	$+ \mid - \mid \ldots$
Expressions	e	::=	$x \mid n \mid x\ op\ x \mid$
			$x[x] \mid \textbf{mux}(x, x, x)$
Statements	s	::=	**skip** $\mid x := e \mid x[x] := x \mid$
			if (x) **then** S **else** $S \mid$
			while (x) **do** $S \mid$
			$x := \textbf{declass}_l(y) \mid$
			$x := \textbf{oram}(y)$
Labeled Statements	S	::=	$l : s \mid S; S$

Fig. 4: Syntax of SCVM

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 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.

Most SCVM 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.

B. Semantics

We define a formal semantics for SCVM programs which we think of as defining a 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-label 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 sometimes find it convenient to project memories whose values 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 $M(x) = (v, l)$, then $v \in$ **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:
$$M = \{x \mapsto (v, l) \mid \begin{array}{l} M_A[\{\texttt{A},\texttt{P}\}](x) = (v, l) \vee \\ M_B[\{\texttt{B}\}](x) = (v, l)\} \end{array}$$
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 communications between $\mathcal{F}$ and each party about the computation are declassifications d_a and d_b (to Alice and Bob, respectively) and notification of termination. 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

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

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

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

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

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

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

Fig. 5: Auxiliary syntax and functions for semantics

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. The trace ϵ indicates an unobservable statement execution (e.g., Bob cannot observe Alice executing her local code). Trace equivalence (i.e. $t_1 \equiv t_2$) is defined 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 value v read or written. Operations on public variables generate memory event **read**(x, v) or **write**(x, v).
- `A`/`B`: Operations on Alice's secret arrays generate memory event **readarr**(x, n, v) or **writearr**(x, n, v) visible to Alice only. Operations on Alice's secret variables generate memory event **read**(x, v) or **write**(x, v) visible to Alice only. Operations on Bob's secret arrays/variables are handled similarly.
- `O`: Operations on a secret array generate memory event x visible to both Alice and Bob, containing only the variable 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 similarly to 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 write $D[A]$ to denote Alice's declassification d_a and $D[B]$ to denote Bob's declassification d_b.

Semantics rules. Now we turn to the semantics, which consists of two judgments. Figure 6 defines rules for the judgment $l \vdash \langle M, e\rangle \Downarrow_{(t_a, t_b)} v$, which states that in mode l, 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. The security label l is passed in by the corresponding assignment statement (i.e. $l : x := e$ or $l : y[x_1] := x_2$). If l is `A` or `B`, then the accesses to public variables are not observable to the other party, whereas if l is `O` then both parties know that an access took place; the $l^\star$ label defined in E-Var and E-Array ensures the proper visibility of such events. Note the E-Array rule uses the $get()$ function to retrieve an element from an array; this function will return a default value 0 if the index is out of bounds. 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 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 generates empty instruction traces and memory traces for both parties regardless of its label. An assignment statement first evaluates the expression to assign, and its trace and the write event constitute the memory trace for this statement. Note that expression is evaluated using the label l of the assignment statement as per the discussion of E-Var and E-Array above.

Declassification $x := \textbf{declass}_l(y)$ declassifies a secret variable y (labeled `O`) to a non-secret variable `x` (not labeled `O`). Both Alice and Bob will observe that y is accessed (as defined by t_a and t_b), whereas the label l of variable x determines who

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

$$\text{E-Var}\ \frac{\begin{array}{c} M(x) = (v, l') \qquad v \in \textbf{Nat} \qquad l' \sqsubseteq l \\ l = \mathtt{0} \Rightarrow l^\star = l' \qquad l \neq \mathtt{0} \Rightarrow l^\star = l \\ (t_a, t_b) = select(l^\star, \textbf{read}(x, v), x) \end{array}}{l \vdash \langle M, x\rangle \Downarrow_{(t_a,t_b)} v}$$

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

$$\text{E-Array}\ \frac{\begin{array}{c} M(x) = (m, l') \qquad m \in \textbf{Array} \qquad l' \sqsubseteq l \\ l = \mathtt{0} \Rightarrow l^\star = l' \qquad l \neq \mathtt{0} \Rightarrow l^\star = l \\ l \vdash \langle M, y\rangle \Downarrow_{(t'_a,t'_b)} v \qquad v' = get(m, v) \\ (t''_a, t''_b) = select(l^\star, \textbf{readarr}(x, v, v'), x) \\ t_a = t'_a@t''_a \qquad t_b = t'_b@t''_b \end{array}}{l \vdash \langle M, x[y]\rangle \Downarrow_{(t_a,t_b)} v'}$$

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

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

sees the declassified value as indicated by the declassification event D.

ORAM initialization produces a shared, secret array x from an array y provided by one party. Thus, the security label of x must be $\mathtt{0}$, and the security label of y must not be $\mathtt{0}$. This rule implies that the party who holds y will observe accesses to y, and then both parties can observe accesses 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 are 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

The ideal functionality $\mathcal{F}$ defines the baseline of security, emulating a trusted third party that runs the program using Alice and Bob's data, directly revealing to them only the explicitly declassified values. In a real implementation run directly by Alice and Bob, however, each party will see additional events of interest, in particular an instruction trace and a memory trace (as defined by the semantics). Importantly, we want to show that these traces provide no additional information about the opposite party's data beyond what each party could learn from observing $\mathcal{F}$. We do this by proving that in fact these traces can be *simulated* by Alice and Bob using their local data and the list of declassification events provided by $\mathcal{F}$. As such, revealing the instruction and memory traces (as in a real implementation) provides no additional useful information.

We call our security property Γ-*simulatability*. To state this property formally, we first define a multi-step version of our statement semantics:

$$\frac{\begin{array}{c} \langle M, P\rangle \xrightarrow{\Gamma,(i_a,t_a,i_b,t_b)}{}^{\star} \langle M_n, P_n\rangle : D_1, ..., D_n \qquad n \geq 0 \\ \langle M_n, P_n\rangle \xrightarrow{(i'_a,t'_a,i'_b,t'_b)} \langle M', P'\rangle : D' \\ D' \neq \epsilon \vee P' = l : \textbf{skip} \qquad M \text{ and } M' \text{ are both } \Gamma\text{-compatible} \end{array}}{\langle M, P\rangle \xrightarrow{\Gamma,(i'_a,t'_a,i'_b,t'_b)}{}^{\star} \langle M', P'\rangle : D_1, ..., D_n, D'}$$

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)**.** *Let Γ be a type environment, and P a program. We say P is* Γ-simulatable *if there exist simulators sim_A and sim_B, which run polynomial time in the data size, 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 $sim_A(M[\{\mathtt{P},\mathtt{A}\}], D_1[A], ..., D_{n-1}[A]) \equiv (i_a, t_a)$ and $sim_B(M[\{\mathtt{P},\mathtt{B}\}], D_1[B], ..., D_{n-1}[B]) \equiv (i_b, t_b)$.*

Intuitively, if P is Γ-simulatable there exists a simulator sim_A that, given public data $M[\{\mathtt{P}\}]$, Alice's secret data $M[\{\mathtt{A}\}]$, and all outputs $D_1[A], ..., D_{n-1}[A]$ declassified to Alice so far, 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, regardless of the values of Bob's secret data.

Note that Γ-simulatability is *termination insensitive*, and information may be leaked based upon whether a program terminates or not [3]. However, as long as all runs of a program are guaranteed to terminate (as is typical for programs run in secure-computation scenarios), no information leakage occurs.

D. Type System

This section presents our type system, which we prove ensures Γ-simulatability. There are two judgments, both defined in Figure 8. The first, written $\Gamma \vdash e : \tau$, states that under environment Γ, expression e evaluates to type τ. The second judgment, written $\Gamma, pc \vdash S$, states that under environment Γ and a *label context* pc, a labeled statement S is type-correct.

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

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

$$\text{S-Declass}\ \frac{\begin{array}{c} M(y) = (v, \mathtt{O}) \qquad l \neq \mathtt{O} \qquad t_a = t_b = y \\ M' = M[x \mapsto (v, l)] \quad i = \mathtt{O} : \textbf{declass}(x, y) \\ D = select(l, (x, v), \epsilon) \end{array}}{\langle M, \mathtt{O} : x := \textbf{declass}_l(y)\rangle \xrightarrow{(i,t_a,i,t_b)} \langle M', \mathtt{O} : \textbf{skip}\rangle : D}$$

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

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

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

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

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

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

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

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

Here, pc is a label that describes the ambient control context; pc is set according to the guards of enclosing conditionals or loops. Note that since a 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` (resp., `B`), then the statement is part of Alice's (resp., Bob's) local code. In general, for a labeled statement $S = l : s$ we enforce the invariant $pc \sqsubseteq l$, and if $pc \neq \mathtt{P}$, then $pc = l$. In so doing, we ensure that if the security label of a statement is `A` (including if-statements and while-statements), then all nested statements also have security label `A`, thus 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 example from 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, 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 setting of secure-computation, padding achieves the same performance as rewriting a secret-branching statement into a **mux** (or a sequence of them). And, using padding would require reasoning about trace patterns, a complication our type system avoids.

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 a 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 property called Γ-*progress* that formalizes the notion of a program that never gets stuck.

Definition 4 (Γ-progress)**.** *Let Γ be a type environment, and let $P = P_0$ be a program. We say P enjoys* Γ-progress

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

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

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

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

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

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

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

$$\text{T-Declass}\frac{pc = \mathtt{P} \quad \Gamma(y) = \textbf{Nat}\ \mathtt{0} \quad \Gamma(x) = \textbf{Nat}\ l \quad l \neq \mathtt{0}}{\Gamma, pc \vdash \mathtt{0} : x := \textbf{declass}_l(y)}$$

$$\text{T-ORAM}\frac{pc = \mathtt{P} \quad \Gamma(x) = \textbf{Array}\ \mathtt{0} \quad \Gamma(y) = \textbf{Array}\ l \quad l \neq \mathtt{0}}{\Gamma, pc \vdash \mathtt{0} : x := \textbf{oram}(y)}$$

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

$$\text{T-Cond}\frac{\Gamma(x) = \textbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{0} \quad pc \neq \mathtt{P} \Rightarrow l = pc \quad \Gamma, l \vdash S_i \quad i = 1,2}{\Gamma, pc \vdash l : \textbf{if}(x)\textbf{then}\ S_1\textbf{else}\ S_2}$$

$$\text{T-While}\frac{\Gamma(x) = \textbf{Nat}\ l \quad pc \sqsubseteq l \quad l \neq \mathtt{0} \quad pc \neq \mathtt{P} \Rightarrow l = pc \quad \Gamma, l \vdash S}{\Gamma, pc \vdash l : \textbf{while}(x)\textbf{do}\ S}$$

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

Fig. 8: Type System for SCVM

if for any Γ*-compatible memories* $M_0, \ldots, M_n$ *for which* $\langle M_j, P_j\rangle \xrightarrow{(i_a^j, t_a^j, i_b^j, t_b^j)} \langle M_{j+1}, P_{j+1}\rangle : D^j$ *for* $j = 0, ..., n-1$*, either* $P_n = l$: ***skip****, or there exist* $i'_a, t'_a, i'_b, t'_b, M', P'$ *such that* $\langle M_n, P_n\rangle \xrightarrow{(i'_a, t'_a, i'_b, t'_b)} \langle M', P'\rangle : D'$.

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

A well-typed program never gets stuck:

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

Proofs of both theorems above can be found in our supplemental technical report [19].

E. From SCVM Programs to Secure Protocols

Let P be a program, and let $\mathcal{F}$ be the ideal functionality based on this program as described earlier. Here we define a *hybrid-world* protocol $\pi^{\mathcal{G}}$ based on P, where $\mathcal{G} = (\mathcal{F}_{\text{op}}, \mathcal{F}_{\text{mux}}, \mathcal{F}_{\text{oram}}, \mathcal{F}_{\text{declass}})$ is a fixed set of ideal functionalities that implement simple binary operations ($\mathcal{F}_{\text{op}}$), a MUX operation ($\mathcal{F}_{\text{mux}}$), ORAM access ($\mathcal{F}_{\text{oram}}$), and declassification ($\mathcal{F}_{\text{declass}}$). Input to each of these ideal functionalities can either be Alice or Bob's local inputs, public inputs, and/or the shares of secret inputs (each share supplied by Alice and Bob respectively). Each ideal functionality is explicitly parameterized by the types of the inputs. Further, except for $\mathcal{F}_{\text{declass}}$ which performs an explicit declassification, all other ideal functionalities return shares of the computation or memory fetch result to Alice and Bob, respectively. Further details of the ideal functionalities are given in our supplemental technical report [19], along with formal definitions of the simulator and hybrid world semantics.

Informally, the hybrid world protocol $\pi^{\mathcal{G}}$ runs as follows:

1) Alice and Bob first agree on public values, ensuring that $M_A[\{\mathtt{P}\}] = M_B[\{\mathtt{P}\}]$. During the protocol each maintains a *declassification list*, for keeping track of previously declassified values, and a *secret memory* that contains shares of secret (non-ORAM) variables. To start, both the lists and memories are empty, i.e., $\mathtt{decls}_A := \mathtt{decls}_B := \epsilon$ and $M_A^S = M_B^S = []$.
2) Alice runs her simulator (locally) on her initial memory to obtain $(i_a, t_a) = sim_A(M_A, \mathtt{decls}_A)$, where i_a and t_a cover the portion of the execution starting from just after the last provided declassification (i.e., the final d_a in the list $\mathtt{decls}_A$) up to the next declassification instruction or the terminating **skip** statement. Bob does likewise to get $(i_b, t_b) = sim_B(M_B, \mathtt{decls}_B)$.
3) Alice executes the instructions in i_a using the hybrid-world semantics, which reads (and writes) secret shares from (to) M_A^S and obtains the values of other reads from events observed in t_a. Bob does similarly with i_b, M_B^S and t_b. The semantics covers three cases:
   - If an instruction in i_a is labeled P, then so is the corresponding instruction in i_b. Both parties execute the instruction.
   - If an instruction in i_a is labeled A, then Alice executes this instruction locally. Bob does similarly for instructions labeled B.
   - If an instruction in i_a is labeled 0, then so is the corresponding instruction in i_b. Alice and Bob call the appropriate ideal-world functionality from $\mathcal{G}$ to execute this instruction. If the instruction is a declassification, then $\mathcal{F}_{\text{declass}}$ will generate an event (d_a, d_b).
4) If the last instruction executed in step 3 is a declassification, then Alice appends her declassification to her local

declassification list (i.e., $\texttt{decls}_A := \texttt{decls}_A\texttt{++}[d_a]$), and Bob does likewise; then both repeat step 2. Otherwise, the protocol completes.

We have proved that if P is Γ-simulatable, then $\pi^{\mathcal{G}}$ securely implements $\mathcal{F}$ against semi-honest adversaries.

Theorem 3. *(Informally) Let P be a program, $\mathcal{F}$ the ideal functionality corresponding to P, and $\pi^{\mathcal{G}}$ the protocol corresponding to P as described above. If P is Γ-simulatable, then $\pi^{\mathcal{G}}$ securely implements $\mathcal{F}$ against semi-honest adversaries in the $\mathcal{G}$-hybrid model.*

Using standard composition results for cryptographic protocols, we obtain as a corollary that if all ideal functionalities in $\mathcal{G}$ are implemented by semi-honest secure protocols, the resulting (real-world) protocol securely implements $\mathcal{F}$ against semi-honest adversaries.

A formal definition of $\pi^{\mathcal{G}}$, formal theorem statement, and a proof of the theorem can be found in our supplemental technical report [19].

V. Compilation

We informally discuss how to compile an annotated C-like source language into a SCVM program. 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 [26] 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) [2], 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 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 is the security label for the array, l_1 is the security label of the index variable and l_2 is the security label of the value variable, the type system should still restrict $l_1 \sqsubseteq l$, but if $l = \texttt{N}$, the type system accepts $l_2 = \texttt{O}$, but requires $l_1 = \texttt{P}$.

Correctness. We do not prove the correctness of our compiler, but instead can use a 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 [7] 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 `0`. 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

```
cond3 := !vis[j] && (bestj<0||dis[j]<bestdis);
bestj := mux(cond3, j, bestj);
bestdis := mux(cond3, dis[j], bestdis);
```

Lines 11-13 are similarly transformed

```
tmp := bestdis + orame[bestj*n+j];
cond4 := !vis[j] && (tmp<dis[j]);
dis[j] := mux(cond4, tmp, dis[j]);
```

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

VI. Evaluation

Programs. We have built several secure two-party computation applications. As run-once tasks, we implemented both the Knuth-Morris-Pratt (KMP) string-matching algorithm as well as Dijkstra's shortest-path algorithm. For repeated sublinear-time database queries, we considered binary search and the heap data structure. All applications are listed in Table II.

Compilation time. All programs took little time (e.g., under 1 second) to compile. In comparison, some earlier circuit-model compilers involve copying datasets into circuits, and therefore the compile-time can be large [16], [21] (e.g., Kreuter et al. [16] report a compilation time of roughly 1000 seconds for an implementation of an algorithm to compute graph isomorphism on 16-node graphs).

In our experiments, we manually checked the correctness of compiled programs (we have not yet implemented a type checker for SCVM, though doing so should be straightforward).

A. Evaluation Methodology

Although our techniques are compatible with any cryptographic back-end secure in the semi-honest model by the definition of Canetti [5], we use the garbled circuit approach in our evaluation [13].

We measure the computational cost by calculating the number of encryptions required by the party running as the circuit generator (the party running as the evaluator does less work). For every non-XOR binary gate, the generator makes 3 block-cipher calls; for every oblivious transfer (OT), 2 block-cipher operations are required since we rely on OT extension [14]. For the *run-once* applications (i.e., Dijkstra shortest distance, KMP-matching, aggregation, inverse permutation), we count in the ORAM initialization cost when comparing to the automated circuit approach (which doesn't require RAM initialization). The ORAM initialization can be done using a Waksman shuffling network [29]. For the applications expecting multiple executions we do not count the ORAM initialization cost since this one-time overhead will be amortized to (nearly) 0 over many executions.

We implemented the binary tree-based ORAM of Shi et al. [27] using garbled circuits, so that array accesses reveal nothing about the (logical) addresses nor the outcomes. Following Gordon et al.'s ORAM encryption technique [10], every block is XOR-shared (i.e., the client stores secret key k while the server stores $(r, f_k(r) \oplus m)$ where f is a family of psuedorandom permutations and m the data block). This adds one 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 block-cipher evaluations as our performance metric. 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 numbers represent bandwidth consumption since every encryption is sent over the network. Therefore, we do not report separately the bandwidth used. Modern processors with AES support can compute 10^8 AES-128 operations per second.

B. Comparison with Automated Circuits

Presently, automated secure computation implementations largely focus on the circuit-model of computation, handling 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 compare our approach with the existing compiled circuits, and demonstrate that our approach scales much better with respect to dataset size.

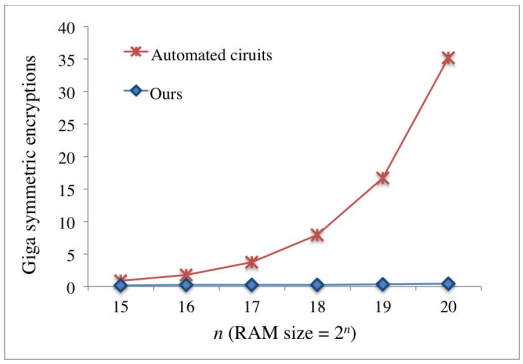
1) Repeated sublinear-time queries: In this scenario, ORAM initialization is a one-time operation whose cost can be amortized over multiple subsequent queries, achieving sublinear amortized cost per query.

Binary search. One example application we tested is binary search, where one party owns a confidential (sorted) array of size n, and the other party searches for (secret) values stored in that array.

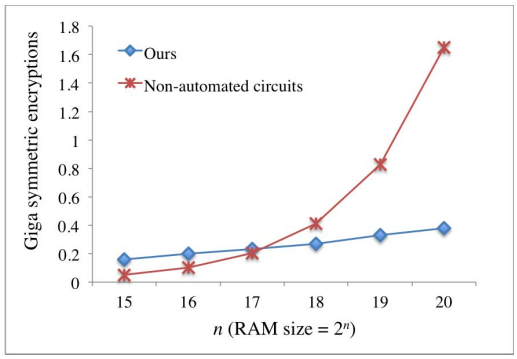
In our experiments, we set the ORAM bucket size to 32 (i.e., each tree-node can store up to 32 blocks). For binary search, we aligned our experimental settings with those of Gordon et al. [10], namely, assuming the size of each data item is 512 bits. We set the recursion factor to 8 (i.e., each block can store up to 8 indices for the data in the upper level recursion tree) and the recursion cut-off threshold to 1000 (namely no more recursion once fewer than 1000 units are to be stored).

Name	Alice's Input	Bob's Input	Setting
Dijkstra's shortest path	a graph	a (src, dest) pair	run-once
Knuth-Morris-Pratt string matching	a sequence	a pattern	run-once
Aggregation over sliding windows	a key-value table	an array of keys	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

TABLE II: Programs used in our evaluation



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



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

Fig. 9: Binary search

Comparing to a circuit-model implementation—which uses a circuit of size $O(n \log n)$ that implements binary search—our approach is faster for all RAM sizes tested (see Figure 9(a)). For $n = 2^{20}$, our approach achieves a $100\times$ speedup.

Note it is also possible to use a smaller circuit of size $O(n)$ that just performs a linear scan over the data. However, such a circuit would have to be "hand-crafted," and would not be output by automated compilation of a binary-search program. Our approach runs faster for large n even when compared to such an implementation (see Figure 9(b)). On data of size $n = 2^{20}$, our approach achieves a $5\times$ speedup even when compared to this "hand-crafted" circuit-based solution.

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 is twice as slow as 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 $100\times$ 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.

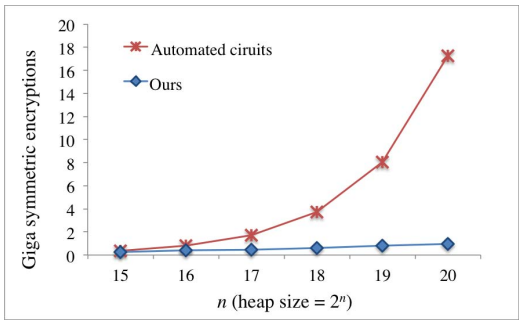
2) Faster one-time executions: We present two applications: the Knuth-Morris-Pratt string-matching algorithm (representative of linear-time RAM programs) and Dijkstra's shortest-path algorithm (representative of super-linear time RAM programs). We compare our approach with a naive program-to-circuit compiler which copies the entire array for every dynamic memory access.

The Knuth-Morris-Pratt algorithm. 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 up to a polylogarithmic factor (due to the ORAM technique).

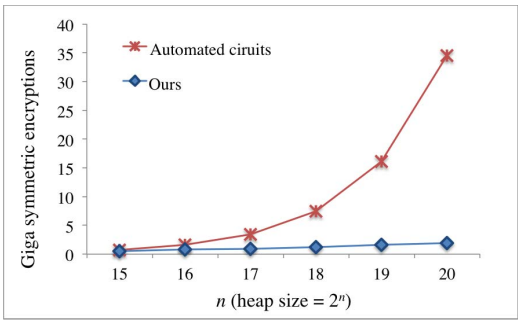
We assume the string T and the pattern P both consist of 16-bit characters. The recursion factor of the ORAM is set to 16. Figure 11(a) and 11(b) show our results compared to those when a circuit-model compiler is used. From Figure 11(a), we can observe that our approach is slower than the circuit-based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

Dijkstra's algorithm. Here Alice has a secret graph while Bob has a secret source/destination pair and wishes to compute the shortest distance between them. Compiling from a standard Dijkstra shortest-path algorithm, we obtain an $O(n^2 \log^3 n)$-overhead RAM-model protocol.

In our experiment, Alice's graph is represented by an $n \times n$

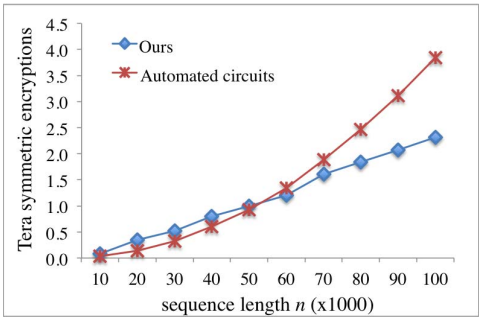


(a) Heap insertion

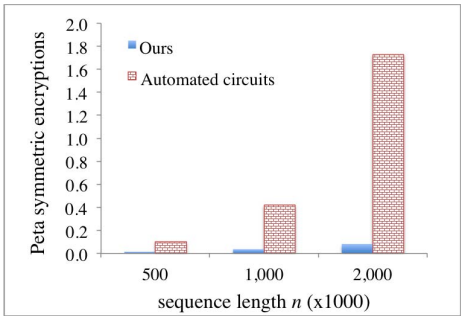


(b) Heap extraction

Fig. 10: Heap operations



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



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

Fig. 11: KMP string matching

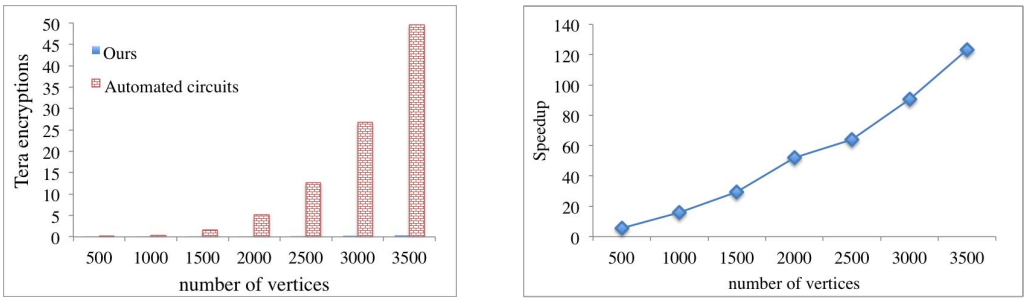
adjacency matrix (of 32-bit integers) where n is the number of vertices in the graph. The distances associated with the edges are denoted by 32-bit integers. We set ORAM recursion factor to 8. The results (Figure 12(a)) show that our scheme runs faster for all sizes of graphs tested. As the performance 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 sliding windows. Alice has a key-value table, and Bob has a (size-n) array of keys. The secure computation task is the following: for every size-k window on the key array, look up k values corresponding to Bob's k keys within the window, and output the minimum value. Our compiler outputs a $O(n \log^3 n)$ 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

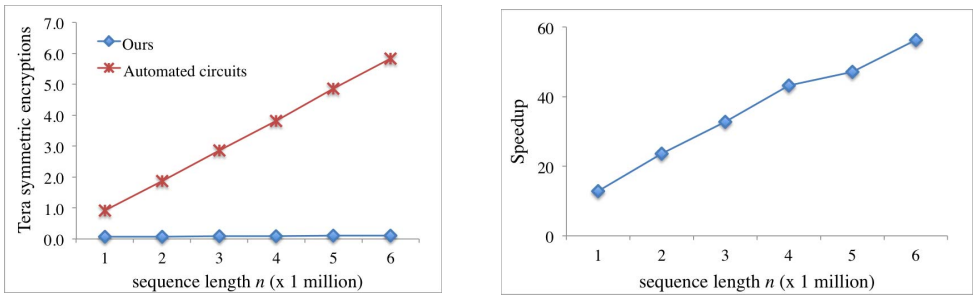
Benefits of instruction-trace obliviousness. The RAM-SC technique of Gordon et al. [10], described in Section II-A, uses a universal next-instruction circuit to hide the program counter and the instructions executed. Each instruction involves ORAM operations for instruction and data fetches, and the next-instruction circuit must effectively execute all possible instructions and use an n-to-1 multiplexer to select the right outcome. Despite the lack of concrete implementation for their general approach, we show through back-of-the-envelope calculations that our approach should be orders-of-magnitude faster.

Consider the problem of binary search over a 1-million item dataset: in each iteration, there are roughly 10 instructions to run, hence 200 instructions in total to complete the search. To run every instruction, a universal-circuit-based implementation has to execute every possible instruction defined in its instruction set. Even if we conservatively assume a RISC-style instruction set, we would require 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).



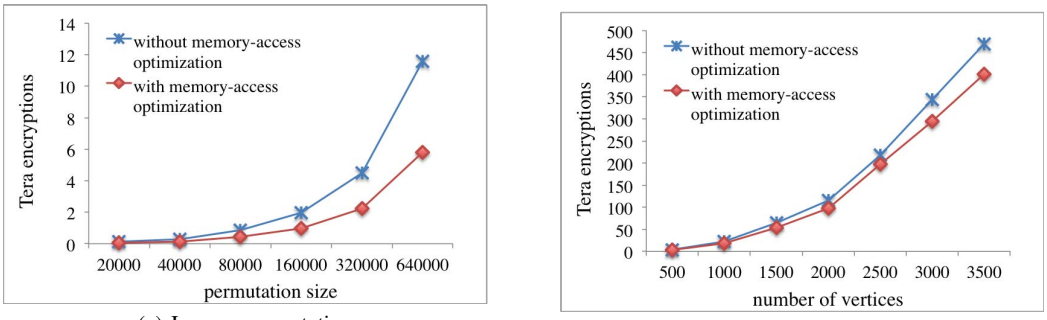
(a) Performance (b) Speedup

Fig. 12: Dijkstra's shortest-path algorithm



(a) Performance (b) Speedup

Fig. 13: Aggregation over sliding windows



(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.

Benefits of memory-trace obliviousness. In addition to avoiding the overhead of a next-instruction circuit, SCVM avoids the overhead of storing all arrays in a single, large ORAM. Instead, SCVM can store some arrays as non-oblivious secret shared memory, and others in separate ORAM banks, rather than one large ORAM. Doing so does not compromise security because the type system ensures memory-trace obliviousness. Here we assess the advantages of these optimizations by comparing against SCVM programs compiled without the optimizations enabled. The results for two applications are given in Figure 14.

- **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 though 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's shortest path.** We discussed the advantages of memory-trace obliviousness in Section III with respect to Dijkstra's algorithm. Our experiments show that we consistently save $15 \sim 20\%$ 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. Conclusions

We describe the first automated approach for RAM-model secure computation. Directions for future work include extending our framework to support malicious security; applying orthogonal techniques (e.g., [7]) to ensure correctness of the compiler; incorporating other cryptographic backends into our framework; and adding additional language features such as higher-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] J. Agat. Transforming out timing leaks. In *POPL*, 2000.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *In POPL*, 1988.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[4] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S & P*, 2013.

[5] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.

[6] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *USENIX Security*, 2013.

[7] COMPCERT: Compilers you can *formally* trust. http://compcert.inria.fr/.

[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 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*, 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*, 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] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. Technical Report CS-TR-5033, University of Maryland, Department of Computer Science, Mar. 2014.

[20] S. Lu and R. Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, 2013.

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

[22] P. Mardziel, M. Hicks, J. Katz, and M. Srivatsa. Knowledge-oriented secure multiparty computation. In *PLAS*, 2012.

[23] 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.

[24] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE S & P*, 2014.

[25] A. Rastogi, P. Mardziel, M. Hammer, and M. Hicks. Knowledge inference for optimizing secure multi-party computation. In *PLAS*, 2013.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[27] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[28] 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 *In CCS*, 2013.

[29] A. Waksman. A permutation network. *J. ACM*, 15, 1968.

[30] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

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

[32] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *S & P*, 2013.