Memory Trace Oblivious Program Execution

Department of Computer Science Technical Report CS-TR-5020

Chang Liu, Michael Hicks, and Elaine Shi
The University of Maryland, College Park, USA

***Abstract*—Cloud computing allows users to delegate data and computation to cloud service providers, at the cost of giving up physical control of their computing infrastructure. An attacker (e.g., insider) with physical access to the computing platform can perform various physical attacks, including probing memory buses and cold-boot style attacks. Previous work on secure (co-)processors provides hardware support for memory encryption and prevents direct leakage of sensitive data over the memory bus. However, an adversary snooping on the bus can still infer sensitive information from the memory access traces. Existing work on Oblivious RAM (ORAM) provides a solution for users to put all data in an ORAM; and accesses to an ORAM are obfuscated such that no information leaks through memory access traces. This method, however, incurs significant memory access overhead.**

In this work, we are among the first to leverage programming language techniques to offer efficient memory-trace oblivious program execution, while providing formal security guarantees. We formally define the notion of memory-trace obliviousness, and provide a type system for verifying that a program satisfies this property. We also describe a compiler that transforms a program into a structurally similar one that satisfies memory trace obliviousness. To achieve optimal efficiency, our compiler partitions variables into several small ORAM banks rather than one large one, without risking security. We use several example programs to demonstrate the efficiency gains our compiler achieves in comparison with the naive method of placing all variables in the same ORAM.

I. Introduction

Cloud computing allows users to delegate their data and computation to computing service providers, and thus relieves users from the necessity to purchase and maintain requisite computing infrastructure. The value proposition is appealing to both cloud providers and clients: market research predicts a 50% compound annual growth rate on public cloud workloads [23].

Despite its increasing popularity, privacy concerns have become a major barrier in furthering cloud adoption. Cloud customers offloading computations transfer both their code and their data to the provider, and thereby relinquish control over both their intellectual property and their private information. While various existing works have considered how to secure sensitive data in the cloud against remote software attacks [15], [31], [48], we consider a stronger adversarial model where the attacker (e.g., a malicious employee of the cloud provider) has physical access to the computing platform. Such an attacker can observe memory contents through cold-boot style attacks [21], [41], malicious peripherals, or by snooping on system buses.

Previous work has proposed the idea of using memory encryption to ensure confidentiality of sensitive memory contents [43], [44], [33], [39], [40]. However, as memory addresses are transferred in cleartext over the memory buses, an adversary can gain sensitive information by observing the memory addresses accessed. For example, address disclosure can leak implicit program execution flows, resulting in the leakage of sensitive code or data [50].

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [17], can be used to protect memory access patterns. In particular, we can place sensitive code and data into ORAM, and doing so has the effect of hiding the access pattern. Roughly speaking, this works by issuing many physical reads/writes for each logical one in the program, and by shuffling the mapping between the logical data and its actual physical location. Unfortunately, placing all code and sensitive data in ORAM leads to significant memory access overhead in practice [42], [47], [12], and can still leak information, e.g., according to the length of the memory trace. On the other hand, customized data-oblivious algorithms have been suggested for specific algorithms, to achieve asymptotically better overhead than generic ORAM simulation [20], [11]. This approach, however, does not scale in terms of human effort.

Contributions. We make four main contributions. First, we define *memory trace obliviousness*, a property that accounts for leaks via the memory access trace. We present a formal semantics for a simple programming language that allocates its secret data and instructions in ORAM, and define the trace of data reads/writes and instruction fetches during execution. We define memory trace obliviousness as an extension of *termination-sensitive noninterference* [2], [34] that accounts for the memory trace as a channel of information. Note that memory trace obliviousness is stronger than the notion used in the traditional ORAM literature [17]—it ensures the content and *length* of the memory access pattern are independent of sensitive inputs, while traditional ORAM security does not provide the latter guarantee.

Second, we present a novel type system for enforcing memory trace obliviousness. This type system builds on standard type systems for checking noninterference [34].

Notably, our type system requires (reminiscent of work by Agat [1] and others) that both branches of a conditional whose guard references secret information produce indistinguishable traces, where trace events consider public and secret data accesses and instruction fetches, and that loop guards not depend on secret information. We also support allocating memory in distinct ORAM banks, for improved performance.

Third, we develop a compilation algorithm for transforming programs that (roughly) satisfy standard noninterference into those that satisfy memory trace obliviousness. There are two distinct problems: allocating data to ORAM banks, and adding padding instructions. Our algorithm employs a solution to the *shortest common supersequence* problem [14] to find common events on the true and false branches of a conditional, and then inserts minimal padding instructions on both sides, so that the traces will be equal.

Finally, we present a small empirical evaluation of our approach with several popular algorithms including Dijkstra's (single-source) shortest path, k-means, matrix multiplication, and find-max. In comparison with generic ORAM simulation (i.e., placing all data in a single ORAM bank), we can always achieve at least a constant factor performance gain. More interestingly, in many cases we can achieve *asymptotic performance gains*. The intuition is that many data mining algorithms traverse data in a predetermined order, independent of the sensitive data contents. For example, the find-max example sequentially scans through a sensitive array to find the maximum. In these cases, it suffices to encrypt the data array in memory, without placing them in ORAM (this is equivalent to placing each element in the array in a separate ORAM bank of size 1).

We present an overview of our approach in the next section. Sections III and IV present our type system and compiler, and Section V presents our empirical evaluation. We compare to related work in Section VI and conclude with ideas for future research in Section VII.

II. Overview

We are interested in the scenario in which an untrusted cloud provider hosts both computation and storage for a client. In particular, the client uploads both code and data to the cloud, and the code is executed over the data on a cloud server. To start, we assume the code to be run is not secret, but that some or all of the uploaded data is. For example, the census bureau does not care if the cloud provider knows which suite of statistical analyses it is running, but does care to keep the census data itself private. While code privacy is not a focus of this paper, we point out that it can easily be achieved by placing all instructions in an ORAM bank dedicated for code – and since code size is typically small in comparison with data size, the performance overhead of doing this is mild in comparison with placing all data in a single ORAM.

Figure 1(a) depicts a motivating example. The program `max` scans through the array `h[]`, and finds its maximum element. The length of the array `n` is labeled `public`, meaning that it may be learned by the adversary. The array `h[]` itself is labeled `secret`, as is the type of the returned value, meaning that the adversary should nothing about either of them. We would like this program to exhibit *termination-sensitive noninterference* when run at the cloud provider—the provider should learn nothing about the contents of `h[]` or the result. We can see that this program would be accepted by a type system for enforcing standard noninterference [34].[1]

A. Threat model

We consider an adversary who has physical access to the cloud server (e.g., a malicious employee of the cloud provider). In particular, we asume the adversary can observe the contents of DRAM, e.g., via cold-boot style attacks. We also assume that the adversary can observe the traffic on the system buses (memory bus, peripheral buses), and can observe when the program terminates. On the other hand, we assume the adversary cannot observe the inner workings of the CPU, i.e., the contents of registers, caches, etc. that are on-chip. In other words, the CPU is trusted. These assumptions roughly correspond to those of the XOM execution model [44], and the software/hardware architecture we present here builds on work in this area.

In addition to sniffing sensitive data, an adversary with physical access could also attempt to tamper with the correct execution of the program. We refer to such attacks as *integrity attacks*. Defending against integrity attacks can be incorporated into our approach using standard techniques such as memory authentication [38], [39], [44], [43], so for simplicity we do not consider them. We assume that there exists a mechanism for the client to securely ship its code and data to (and from) the cloud provider and start execution (more on this below).

Though they are a real threat, in this paper we do not consider timing and other covert channels. We believe we can incorporate ideas from related research to handle timing leaks [1], [10], [26]. Preventing/mitigating timing leaks is neither necessary nor sufficient for preventing leaks due to observed memory traffic, which is our focus in this paper; see Section VI for further discussion. In reality, processor optimizations such as caching and branch prediction can affect what visible memory traces are generated during program execution. In this paper, we assume no caching or branch prediction – how to allow such chip-level features while ensuring memory trace obliviousness is left as future work and discussed in Section VII.

[1] The typing of arrays is slightly non-standard: as explained in the next section, we permit public indexes to secret arrays by using a "failure-oblivious" semantics that ignores out-of-bounds accesses; this semantics resembles Deng and Smith's "lenient" semantics [9].

```
1 secret int max( public int n,
2                 secret int h[]) {
3   public int i = 0;
4   secret int m = 0;
5   while (i < n) {
6     if (h[i] > m) then m = h[i];
7     i++; }
8   return m;
9 }
```

(a) original program

```
1 secret1 int max( public int n,
2                  secret2 int h[]) {
3   public int i = 0;
4   secret1 int m = 0, mdummy = 0;
5   while (i < n) {
6     if (h[i] > m) then m = h[i];3
7     else mdummy = h[i];3
8     i++; }
9   return m; }
```

(b) transformed program

Figure 1. Example program and its memory-oblivious transformation.

B. Approach

We now detail our approach step by step.

Encrypting secrets. To hide data from an adversary who can inspect the cloud server's storage—DRAM in particular—we can use encryption. That is, any secrets can be stored in memory in encrypted form, then decrypted when computed with, and encrypted again before storing the results back to memory. Given that we only trust the CPU, we require that the encryption/decryption be performed entirely on chip (rather than in software) and we need a way to securely load the encryption key onto the chip without revealing it to the adversary. On-chip encryption is now a standard feature (AES has been supported on Intel chips since 2010), and we can use code attestation [30], [37], [35] to achieve secure and verifiable bootstrapping of the program and the encryption key. While existing trusted computing and code attestation are not bullet proof against physical attacks, hardware security modules on chip [24] are starting to attract attention and can provide the needed security.

Even with these measures in place, the adversary can still observe the stream of accesses to memory, even if he cannot observe the content of those accesses, and such observations are sufficient to infer secret information. For our example, we see that the conditional on line 6 will produce a non-zero number of events when the guard is true, but no events when it is false. As such, by observing the trace the adversary could learn the index of the maximum value. Prior work has also observed that the memory trace can leak sensitive information [50], [44], [43].

To eliminate this channel of information, we need a way to run the program so that the event stream does not depend on the secret data—no matter the values of the secret, the observable events will be the same. Programs that exhibit this behavior enjoy a property we call *memory trace obliviousness.*

Padding. Toward ensuring memory trace obliviousness, the compiler can add padding instructions to either or both branches of `if` statements whose guards reference secret information (we refer to such guards as *high guards*). This idea is similar to inserting padding to ensure uniform timing [1], [4], [22], [7]. Looking at Figure 1(b) we can see the original program transformed to add an `else` branch on line 8 that aims to produce the same events as the `if` branch. Unfortunately, this approach does not quite work because while number and kind of events in the added branch is the same, the write address is different—for the true branch the program writes to `m` while for the false branch it writes to `mdummy`. We need to somehow hide the addresses being read from/written to.

ORAM for secret data. We can solve this problem by storing secret data in Oblivious RAM (ORAM), and extend our trusted computing base with an on-chip ORAM controller. This controller will encrypt/decrypt the secret data and maintain a mapping between addresses for variables used by the program and actual storage addresses for those variables in DRAM. For each read/write issued for a secret program address the ORAM controller will issue a series of reads/writes to actual DRAM addresses, which has the effect of hiding which of the accesses was the real address. Moreover, with each access, the ORAM controller will shuffle program/storage address mappings so that the physical location of any program variable is constantly in flux. Asymptotically, ORAM accesses are polylogarithmic in the size of the ORAM [17]. Note that if we were concerned about integrity, we could compose the ORAM controller with machinery for, say, authenticated accesses.[2]

Returning to the example, we can see that by allocating all secret data in ORAM, and `mdummy` and `m` in particular, we ensure that both branches produce indistinguishable traces, consisting of: read of `i`, ORAM event (for the read of `h[i]`), ORAM event (for the read of `m`), read of `i`, ORAM event (for the read of `h[i]`), and ORAM event (for the write to `m` or `mdummy`).

Note that loops can also be source of trace information: if the guard is secret, then the number of loop iterations (as reflected in the trace) could reveal something about the secret. Therefore we forbid loops with secret guards, and forbid loops of any kind in conditionals. Fortunately, this is not too onerous in the common case that inputs and outputs are secret, but the length of secret data (or an upper bound

[2] A detailed hardware design is outside the scope of this work, but we are indeed in the process of constructing one [29].

on that length) can be known.

Storing (some) code in ORAM. The careful reader will have observed that while the *data* accesses of the two branches now produce indistinguishable traces, the *instruction* fetches can be distinguished: depending on whether `h[i] > m` we will either fetch instructions corresponding to statement 7 or statement 8. Since instructions are stored in unencrypted DRAM, the adversary can observe them being fetched. We can solve this problem by storing some instructions in ORAM so as to effectively hide the program counter; in general we must store instructions on both branches of a conditional with a high guard. In our example, we illustrate this fact by drawing a box around the affected statements on lines 7 and 8.

Multiple ORAM banks. ORAM can be an order of magnitude slower than regular DRAM [42]. Moreover, larger ORAM banks containing more variables incur higher overhead than smaller ORAM banks [17], [36]; as mentioned above, ORAM accesses are asymptotically related to the size of the ORAM. Thus we can reduce run-time overhead by allocating code/data in multiple, smaller ORAM banks rather than all of it in a single, large bank.

We observe that for the purposes of memory trace obliviousness, we do not need to group all secret addresses in the same ORAM bank. For our example, we only need to make accesses to `m` and `mdummy` indistinguishable, and fetches from the boxed statements on lines 7 and 8; we do not need to differentiate a fetch from line 7 from a read/write of `m`. In particular, we can use three distinct ORAM banks, which are indicated by subscripts on `secret` qualifiers in the figure: `m` and `mdummy` go in one bank, `h` goes in another, and code on lines 7 and 8 goes in a third.

Arrays. Implicitly we have assumed that all of an array is allocated to the same ORAM bank, but this need not be the case. Indeed, for our example it is safe to simply encrypt the contents of `h[i]` because knowing which memory address we are accessing does not happen to reveal anything about the contents of `h[]`. This is because the access pattern on the array does not depend on any secret—every execution of `max` will access the same array elements in the same order.

If we allocate each array element in a separate ORAM bank, the running time of the program becomes roughly $6n$ accesses: each access to `h[i]` is in a bank of size 1, and `m` or `mdummy` are in a bank of size 2. In both cases we can access these variables securely using the "trivial ORAM," which simply scans every element in its bank; thus there is one access for each read of `h[]` and two accesses for each read/write of `m` and `mdummy`, for a total of $6n$ accesses.

In comparison, the naïve strategy of allocating all variables in a single ORAM bank would incur $4n \cdot \text{poly}\log(n+2))$ memory accesses (for secret variables), since each access to an ORAM bank of size m requires $O(\text{poly}\log(m))$ physical memory accesses. This shows that we can achieve asymptotic gains in performance for some programs.

$$
\begin{array}{llcl}
\text{Variables} & x, y, z & \in & \textbf{Vars} \\
\text{Numbers} & n & \in & \textbf{Nat} \\
\text{ORAM bank} & o & \in & \textbf{ORAMbanks} \\
\text{Expressions} & e & ::= & x \mid e\ op\ e \mid x[e] \mid n \\
\text{Statements} & s & ::= & \textbf{skip} \mid x := e \mid x[e] := e \mid \\
 & & & \textbf{if}(e, S, S) \mid \textbf{while}\ (e, S) \\
\text{Programs} & S & ::= & p{:}s \mid S; S \\
\text{Locations} & p & ::= & n \mid o
\end{array}
$$

Figure 2. Language syntax

$$
\begin{array}{llcl}
\text{Arrays} & m & \in & \textbf{Arrays} = \textbf{Nat} \rightharpoonup \textbf{Nat} \\
\text{Labels} & l & \in & \textbf{SecLabels} = \{L\} \cup \textbf{ORAMbanks} \\
\text{Memory} & M & \in & \textbf{Vars} \rightharpoonup (\textbf{Arrays} \cup \textbf{Nat}) \times \textbf{SecLabels} \\
\text{Traces} & t & ::= & \textbf{read}(x, n) \mid \textbf{readarr}(x, n, n) \mid \\
 & & & \textbf{write}(x, n) \mid \textbf{writearr}(x, n, n) \mid \\
 & & & \textbf{fetch}(p) \mid o \mid t@t \mid \epsilon
\end{array}
$$

$$
get(m, n) = \begin{cases} m(n) & \text{if } 0 \leq n < |m| \\ 0 & \text{otherwise} \end{cases}
$$

$$
upd(m, n_1, n_2) = \begin{cases} m[n_1 \mapsto n_2] & \text{if } 0 \leq n_1 < |m| \\ m & \text{otherwise} \end{cases}
$$

$$
evt(l, t) = \begin{cases} l & \text{if } l \in \textbf{ORAMbanks} \\ t & \text{otherwise} \end{cases}
$$

Figure 3. Auxiliary syntax and functions for semantics

III. Memory trace obliviousness by typing

This section formalizes a type system for verifying that programs like the one in Figure 1(b) enjoy memory trace obliviousness. In the next section we describe a compiler to transform programs like the one in Figure 1(a) so they can be verified by our type system.

We formalize our type system using the simple language presented in Figure 2. Programs S consist of a sequence $S; S$ of *labeled statements* $p{:}s$, where p is either a number n unique to the program (i.e., a line number) or is an ORAM bank identifier o; in the latter case, the statement is stored in the corresponding ORAM bank, while in the former it is stored in unencrypted RAM. Statements s include the no-op **skip**, assignments to variables and arrays, conditionals, and loops. Expressions e consist of constant natural numbers, variable and array reads, and (compound) operations. For simplicity, arrays may contain only integers (and not other arrays), and bulk assignments between arrays (i.e., $x := y$ when y is an array) are not permitted.

A. Operational semantics

We define a big-step operational semantics for our language in Figure 4, which refers to auxiliary functions and

$$\boxed{\langle M, e\rangle \Downarrow_t n}$$

$$\text{E-Var}\ \frac{\begin{array}{c} M(x) = (n, l) \\ t = \mathit{evt}(l, \textbf{read}(x, n)) \end{array}}{\langle M, x\rangle \Downarrow_t n} \qquad \text{E-Const}\ \frac{}{\langle M, n\rangle \Downarrow_\epsilon n}$$

$$\text{E-Op}\ \frac{\langle M, e_1\rangle \Downarrow_{t_1} n_1 \quad \langle M, e_2\rangle \Downarrow_{t_2} n_2 \quad n = n_1\ \mathit{op}\ n_2}{\langle M, e_1\ \mathit{op}\ e_2\rangle \Downarrow_{t_1 @ t_2} n}$$

$$\text{E-Arr}\ \frac{\begin{array}{c} \langle M, e\rangle \Downarrow_t n \quad M(x) = (m, l) \quad n' = \mathit{get}(m, n) \\ t_1 = \mathit{evt}(l, \textbf{readarr}(x, n, n')) \end{array}}{\langle M, x[e]\rangle \Downarrow_{t @ t_1} n'}$$

$$\boxed{\langle M, s\rangle \Downarrow_t M'}$$

$$\text{S-Skip}\ \frac{}{\langle M, \textbf{skip}\rangle \Downarrow_\epsilon M}$$

$$\text{S-Asn}\ \frac{\langle M, e\rangle \Downarrow_t n \quad M(x) = (n', l) \quad t' = \mathit{evt}(l, \textbf{write}(x, n))}{\langle M, x := e\rangle \Downarrow_{t @ t'} M[x \mapsto (n, l)]}$$

$$\text{S-AAsn}\ \frac{\begin{array}{c} \langle M, e_1\rangle \Downarrow_{t_1} n_1 \quad \langle M, e_2\rangle \Downarrow_{t_2} n_2 \quad M(x) = (m, l) \\ m' = \mathit{upd}(m, n_1, n_2) \quad t = \mathit{evt}(l, \textbf{writearr}(x, n_1, n_2)) \end{array}}{\langle M, x[e_1] := e_2\rangle \Downarrow_{t_1 @ t_2 @ t} M[x \mapsto (m', l)]}$$

$$\text{S-Cond}\ \frac{\langle M, e\rangle \Downarrow_{t_1} n \quad \langle M, S_i\rangle \Downarrow_{t_2} M \ \ (i = \mathit{ite}(n, 1, 2))}{\langle M, \textbf{if}\ (e, S_1, S_2)\ \rangle \Downarrow_{t_1 @ t_2} M'}$$

$$\text{S-WhileT}\ \frac{\begin{array}{c} \langle M, e\rangle \Downarrow_t n \qquad n \neq 0 \\ \langle M, (S; p{:}\textbf{while}(e, S))\rangle \Downarrow_{t'} M' \end{array}}{\langle M, p{:}\textbf{while}(e, S)\rangle \Downarrow_{\textbf{fetch}(p) @ t @ t'} M'}$$

$$\text{S-WhileF}\ \frac{\langle M, e\rangle \Downarrow_t 0}{\langle M, p{:}\textbf{while}(e, S)\rangle \Downarrow_{\textbf{fetch}(p) @ t} M}$$

$$\boxed{\langle M, S\rangle \Downarrow_t M'}$$

$$\text{P-Stmt}\ \frac{\langle M, s\rangle \Downarrow_t M' \quad t' = \textbf{fetch}(p)}{\langle M, p{:}s\rangle \Downarrow_{t' @ t} M'}$$

$$\text{P-Stmts}\ \frac{\langle M, S_1\rangle \Downarrow_{t_1} M' \quad \langle M', S_2\rangle \Downarrow_{t_2} M''}{\langle M, S_1; S_2\rangle \Downarrow_{t_1 @ t_2} M''}$$

Figure 4. Operational semantics

syntax defined in Figure 3. Big-step semantics is simpler than the small-step alternative, and though it cannot be used to reason about non-terminating programs, our cloud computing scenario generally assumes that programs terminate. The main judgment of the former figure, $\langle M, S\rangle \Downarrow_t M'$ (shown at the bottom), indicates that program S when run under memory M will terminate with new memory M' and in the process produce a *memory access trace* t. We also define judgments $\langle M, s\rangle \Downarrow_t M'$ and $\langle M, e\rangle \Downarrow_t n$ for evaluating statements and expressions, respectively.

We model memories M as partial functions from variables to labeled values, where a value is either an array m or a number n, and a label is either L or an ORAM bank identifier o. Thus we can think of an ORAM bank o containing all data for variables x such that $M(x) = (\_, o)$, whereas all data labeled L is stored in normal RAM. We model an array m as a partial function from natural numbers to natural numbers. We write $|m|$ to model the length of the array; that is, if $|m| = n$ then $m(i)$ is defined for $0 \leq i < n$ but nothing else. To keep the formalism simple, we assume all of the data in an array is stored in the same place, i.e., all in RAM or all in the same ORAM bank. We sketch how to relax this assumption, to further improve performance, in Section III-E.

A memory access trace t is a finite sequence of events arising during program execution that are observable by tapping the memory bus. These events include read events $\textbf{read}(x, n)$ which states that number n was read from variable x and $\textbf{read}(x, n_1, n_2)$, which states number n_2 was read from $x[n_1]$. The corresponding events for writes to variables and arrays are similar, but refer to the number written, rather than read. Event $\textbf{fetch}(p)$ indicates a fetch of the statement at location p in the program. Recall that p could be either an ORAM bank or a unique number, where the former reflects that the particular instruction is unknown (since it is stored in ORAM) while the latter indicates the precise statement number stored in RAM. Event o indicates an access to ORAM—only the storage bank o is discernable, not the precise variable involved or even whether the access is a read or a write. (Each ORAM read/write in the program translates to several actual DRAM accesses, but we model them as a single abstract event.) Finally, $t@t$ represents the concatenation of two traces and ϵ is the empty trace.

The rules in Figure 4 are largely straightforward. Rule (E-Var) defines variable reads by looking up the variable in memory, and then emitting an event consonant with the label on the variable's memory. This is done using the evt function defined in Figure 3: if the label is some ORAM bank o then event o will be emitted, otherwise event $\textbf{read}(x, n)$ is emitted since the access is to normal RAM.

The semantics treats array accesses as "oblivious" to avoid information leakage due to out-of-bounds indexes. In particular, rule (E-Arr) indexes the array using auxiliary function get, also defined in Figure 3, that returns 0 if the index n is out of bounds. Rule (S-AAsn) uses the upd function similarly: if the write is out of bounds, then the array is not affected.[3] We could have defined the semantics to throw an exception, or result in a stuck execution, but this would add unnecessary complication. Supposing we had such exceptions, our semantics models wrapping array reads

[3] The syntax $m[n_1 \mapsto n_2]$ defines a partial function m' such that $m'(n_1) = n_2$ and otherwise $m'(n) = m(n)$ when $n \neq n_1$. We use the same syntax for updating memories M.

and writes with a try-catch block that ignores the exception, which is a common pattern, e.g., in Jif [25], [5], and has also been advocated by Deng and Smith [9].

The rule (S-Cond) for conditionals is the obvious one; we write *ite(x,y,z)* to denote y when x is 0, and z otherwise. Rule (S-WhileT) expands the loop one unrolling when the guard is true and evaluates that to the final memory, and rule (S-WhileF) does nothing when the guard is false. Notice that the expanded loop in the premise has the same label p as the original. For statements other than loops, the rule (P-Stmt) factors out the handling of the location label: it issues a fetch event according to the location label p, followed by the trace resulting from evaluating the statement s itself. Finally rule (P-Stmts) handles sequences of statements.

B. Memory trace obliviousness

The security property of interest in our setting we call *memory trace obliviousness*. This property generalizes the standard (termination-sensitive) noninterference property to account for memory traces. Intuitively, a program satisfies memory trace obliviousness if it will always generate the same trace (and the same final memory) for the same adversary-visible memories, no matter the particular values stored in ORAM. We formalize the property in three steps. First we define what it means for two memories to be low-equivalent, which holds when they agree on memory contents having label L.

Definition 1 (Low equivalence)**.** *Two memories M_1 and M_2 are* low-equivalent*, denoted as $M_1 \sim_L M_2$, if and only if $\forall x, v. M_1(x) = (v, L) \Leftrightarrow M_2(x) = (v, L)$.*

Next, we define the notion of the Γ*-validity* of a memory M. Here, Γ is the *type environment* that maps variables to security types τ, which are either $\mathrm{Nat}\ l$ or $\mathrm{Array}\ l$ (both are defined in Figure 5). In essence, Γ indicates a mapping of variables to memory banks, and if memory M employs that mapping then it is valid with respect to Γ.

Definition 2 (Γ-validity)**.** *A memory M is valid under a environment Γ, or* Γ-valid*, if and only if, for all x*

$$\Gamma(x) = \mathit{Nat}\ l \Leftrightarrow \exists n \in \textbf{\textit{Nat}}. M(x) = (n, l)$$
$$\Gamma(x) = \mathit{Array}\ l \Leftrightarrow \exists m \in \textbf{\textit{Arrays}}. M(x) = (m, l)$$

Finally, we define memory trace obliviousness. Intuitively, a program enjoys this property if all runs of the program on low-equivalent, Γ-valid memories will always produce the same trace and low-equivalent final memory.

Definition 3 (Memory trace obliviousness)**.** *Given a security environment Γ, a program S satisfies* Γ-memory trace obliviousness *if for any two Γ-valid memories $M_1 \sim_L M_2$, if $\langle M_1, S\rangle \Downarrow_{t_1} M_1'$ and $\langle M_2, S\rangle \Downarrow_{t_2} M_2'$, then $t_1 \equiv t_2$, and $M_1' \sim_L M_2'$.*

In this definition, we write $t_1 \equiv t_2$ to denote that t_1 and t_2 are equivalent. Equivalence is defined formally in Appendix I, Figure 10. Intuitively, two traces are equivalent if they are syntactically equivalent or we can apply associativity to transform one into the other. Furthermore, ϵ plays the role of the identity element.

$$\begin{array}{lll}
\text{Types} & \tau ::= & \mathrm{Int}\ l \mid \mathrm{Array}\ l \\
\text{Environments} & \Gamma \ \in & \textbf{Vars} \rightharpoonup \textbf{Types} \\
\text{Trace Pats} & T ::= & \textbf{Read}\ x \mid \textbf{Write}\ x \mid \\
 & & \textbf{Readarr}\ x \mid \textbf{Writearr}\ x \mid \\
 & & \textbf{Loop}(p, T, T) \mid \textbf{Fetch}(p) \mid o \mid \\
 & & T@T \mid T + T \mid \epsilon
\end{array}$$

$$l_1 \sqcup l_2 \quad = \quad \begin{cases} l_1 & \text{if } l_1 \neq \mathrm{L} \\ l_2 & \text{otherwise} \end{cases}$$

$$l_1 \sqsubseteq l_2 \quad \mathit{iff} \quad \begin{array}{l} l_1 = L \text{ or} \\ l_2 \neq L \text{ and } l_2 \neq n \end{array}$$

$$\mathit{select}(T_1, T_2) \quad = \quad \begin{cases} T_1 & \text{if } T_1 \sim_L T_2 \\ T_1 + T_2 & \text{otherwise} \end{cases}$$

Figure 5. Auxiliary syntax and functions for typing

C. Security typing

Figure 6 presents a type system that aims to ensure memory trace obliviousness. Auxiliary definitions used in the type rules are given in Figure 5. This type system borrows ideas from standard security type systems that aim to enforce (traditional) noninterference. For the purposes of typing, we define a lattice ordering $\sqsubseteq$ among security labels l as shown in Figure 5, which also shows the $\sqcup$ (join) operation. Essentially, these definitions treat all ORAM bank labels o as equivalent for the purposes of typing (you can think of them as the "high" security label). In the definition of $\sqsubseteq$, we also consider the case when l_2 could be a program location n, which is treated as equivalent to L (this case comes up when typing labeled statements).

The typing judgment for expressions is written $\Gamma \vdash e : \tau; T$, which states that in environment Γ, expression e has type τ, and when evaluated will produce a trace described by the *trace pattern* T. The judgments for statements s and programs S are similar. Trace patterns describe families of run-time traces; we write $t \in T$ to say that trace t matches the trace pattern T.

Trace pattern elements are quite similar to their trace counterparts: fetches and ORAM accesses are the same, as are empty traces and trace concatenation. Trace pattern events for reads and writes to variables and arrays are more abstract, mentioning the variable being read, and not the particular value (or index, in the case of arrays); we have $\textbf{read}(x, n) \in \textbf{Read}(x)$ for all n, for example. There is also the *or*-pattern $T_1 + T_2$ which matches traces t such that either $t \in T_1$ or $t \in T_2$. Finally, the trace pattern for loops, $\textbf{Loop}(p, T_1, T_2)$, denotes the set of patterns $\textbf{Fetch}(p)@T_1$ and $\textbf{Fetch}(p)@T_1@T_2@\textbf{Fetch}(p)@T_1$ and

$$\boxed{\Gamma \vdash e : \tau; T}$$

$$\text{T-Var}\ \frac{\begin{array}{c}\Gamma(x) = \text{Nat } l \\ T = evt(l, \textbf{Read}(x))\end{array}}{\Gamma \vdash x : \text{Nat } l; T} \qquad \text{T-Con}\ \frac{}{\Gamma \vdash n : \text{Nat L}; \epsilon}$$

$$\text{T-Op}\ \frac{\Gamma \vdash e_1 : \text{Nat } l_1; T_1 \quad \Gamma \vdash e_2 : \text{Nat } l_2; T_2 \quad l = l_1 \sqcup l_2}{\Gamma \vdash e_1\ op\ e_2 : \text{Nat } l; T_1 @ T_2}$$

$$\text{T-Arr}\ \frac{\begin{array}{c}\Gamma(x) = \text{Array } l \quad \Gamma \vdash e : \text{Nat } l'; T \quad l' \sqsubseteq l \\ T' = evt(l, \textbf{Readarr}(x))\end{array}}{\Gamma \vdash x[e] : \text{Nat } l \sqcup l'; T @ T'}$$

$$\boxed{\Gamma, l \vdash s; T}$$

$$\text{T-Skip}\ \frac{}{\Gamma, l_0 \vdash \textbf{skip}; \epsilon}$$

$$\text{T-Asn}\ \frac{\Gamma \vdash e : \text{Nat } l; T \quad \Gamma(x) = \text{Nat } l' \quad l_0 \sqcup l \sqsubseteq l'}{\Gamma, l_0 \vdash x := e; T @ evt(l', \textbf{Write}(x))}$$

$$\text{T-AAsn}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \text{Nat } l_1; T_1 \quad \Gamma \vdash e_2 : \text{Nat } l_2; T_2 \\ \Gamma(x) = \text{Array } l \quad l_1 \sqcup l_2 \sqcup l_0 \sqsubseteq l\end{array}}{\Gamma, l_0 \vdash x[e_1] := e_2; T_1 @ T_2 @ evt(l, \textbf{Writearr}(x))}$$

$$\text{T-Cond}\ \frac{\begin{array}{c}\Gamma \vdash e : \text{Nat } l; T \quad \Gamma, l \sqcup l_0 \vdash S_i; T_i\ (i = 1, 2) \\ l \sqcup l_0 \neq \text{L} \Rightarrow T_1 \sim_L T_2 \quad T' = select(T_1, T_2)\end{array}}{\Gamma, l_0 \vdash \textbf{if}(e, S_1, S_2); T @ T'}$$

$$\text{T-While}\ \frac{\Gamma \vdash e : \text{Nat } l; T_1 \quad \Gamma, l_0 \vdash S; T_2 \quad l \sqcup l_0 \sqsubseteq \text{L}}{\Gamma, l_0 \vdash p{:}\textbf{while}(e, S); \textbf{Loop}(p, T_1, T_2)}$$

$$\boxed{\Gamma, l \vdash S; T}$$

$$\text{T-Lab}\ \frac{\Gamma, l_0 \vdash s; T \quad l_0 \sqsubseteq p}{\Gamma, l_0 \vdash p{:}s; (\textbf{Fetch}\ p) @ T}$$

$$\text{T-Seq}\ \frac{\Gamma, l_0 \vdash S_1; T_1 \quad \Gamma, l_0 \vdash S_2; T_2}{\Gamma, l_0 \vdash S_1; S_2; T_1 @ T_2}$$

Figure 6. Typing

$\textbf{Fetch}(p)@T_1@T_2@\textbf{Fetch}(p)@T_1@T_2@\textbf{Fetch}(p)@T_1$ and so on, and thus matches any trace that matches one of them.

Turning to the rules, we can see that each one is structurally similar to the corresponding semantics rule. Each rule likewise uses the evt function (Figure 3) to selectively generate an ORAM event o or a basic event, depending on the label of the variable being read/written. Rule (T-Var) thus generates a $\textbf{Read}(x)$ pattern if x's label is L, or generates the ORAM event l (where $l \neq L$ implies l is some bank o). As expected, constants n are labeled L by (T-Con), and compound expressions are labeled with the join of the labels of the respective sub-expressions by (T-Op). Rule (T-Arr) is interesting in that we require $l \sqsubseteq l'$, where l is the label of the index and l' is the label of the array, but the label of the resulting expression is the join of the two. As such, we can have a public index of a secret array, but not the other way around. This is permitted because of our oblivious semantics: a public index reveals nothing about the length of the array when the returned result is secret, and no out-of-bounds exception is possible.

The judgment for statements $\Gamma, l_0 \vdash s; T$ is similar to the judgment for expressions, but there is no final type, and it employs the standard *program counter (PC) label* l_0 to prevent implicit flows. In particular, the (T-Asn) and (T-AAsn) rules both require that the join of the label l of the expression on the rhs, when joined with the program counter label l_0, must be lower than or equal to the label l' of the variable; with arrays, we must also join with the label l_1 of the indexing expression. Rule (T-Cond) checks the statements S_i under the program counter label that is at least as high as the label of the guard. As such, coupled with the constraints on assignments, any branch on a high-security expression will not leak information about that expression via an assignment to a low-security variable. In a similar way, rule (T-Lab) requires that the statement location p is lower or equal to the program counter label, so that a public instruction fetch cannot be the source of an implicit flow.

Rule (T-Cond) also ensures that if the PC label or that of the guard expression is secret, then the actual run-time trace of the true branch (matched by the trace pattern T_1) and the false branch (pattern T_2) must be equal; if they were not, then the difference would reveal something about the respective guard. We ensure run-time traces will be equal by requiring the trace patterns T_1 and T_2 are *equivalent*, as axiomatized in Figure 7. The first two rows prove that ϵ is the identity, that $\sim_L$ is a transitive relation, and that concatenation is associative. The third row unsurprisingly proves that ORAM events to the same bank and fetches of the same location/bank are equivalent. More interestingly, the third row claims that public reads to the same variable are equivalent. This makes sense given that public writes are *not* equivalent. As such, reads in both branches will always return the same run-time value they had prior to the conditional. Notice that the public reads to the same arrays are also *not* equivalent, since indices may leak information. Finally, the (T-Cond) emits trace T', which according to the *select* function (Figure 5) will be T_1 when the two are equivalent. As such, conditionals in a high context will never produce or-pattern traces (which are not equivalent to any other trace pattern).

Rule (T-While) requires, by the constraint $l \sqcup l_0 \sqsubseteq L$, requires that loop guards be public (which is why we need not join l_0 with l when checking the body S). This constraint ensures that the length of the trace as related to the number of loop iterations cannot reveal something about secret data. Fortunately, this restriction is not problematic for many examples because secret arrays can be safely indexed

$$\frac{T \sim_L T}{\epsilon @ T \sim_L T @ \epsilon \sim_L T} \qquad \epsilon \sim_L \epsilon \qquad \frac{T_1 \sim_L T_2 \quad T_2 \sim_L T_3}{T_1 \sim_L T_3}$$

$$\frac{T_1 \sim_L T_1' \quad T_2 \sim_L T_2'}{T_1 @ T_2 \sim_L T_1' @ T_2'} \qquad (T_1 @ T_2) @ T_3 \sim_L T_1 @ (T_2 @ T_3)$$

$$o \sim_L o \qquad \textbf{Fetch}\ p \sim_L \textbf{Fetch}\ p \qquad \textbf{Read}\ x \sim_L \textbf{Read}\ x$$

Figure 7. Trace pattern equivalence

by public values, and thus looping over arrays reveals no information about them.

Finally, we can prove that well typed programs enjoy memory trace obliviousness.

Theorem 1. *If* $\Gamma, l \vdash S; T$*, then* S *satisfies memory trace obliviousness.*

The full proof can be found in Appendix I.

D. Examples

Now we consider a few programs that do and do not type check in our system. In the examples, public (low security) variables begin with `p`, and secret (high security) variables begin with `s`; we assume each secret variable is allocated in its own ORAM bank (and ignore statement labels).

There are some interesting differences in our type system and standard information flow type systems. One is that we prohibit low reads under high guards that could differ in both branches. For example, the program `if s > 0 then s :=` $\mathtt{p}_1$ `else s :=` $\mathtt{p}_2$ is accepted in the standard type system but rejected in ours. This is because in our system we allow the adversary to observe public reads, and thus he can distinguish the two branches, whereas an adversary can only observe public writes in the standard noninterference proof. On the other hand, the program `if s > 0 then s := p+1 else s := p+2` would be accepted, because both branches will exhibit the same trace

Another difference is that we do not allow high guards in loops, so a program like the following is acceptable in the standard type system is rejected in ours:

```
 s := slen; sum := 0;
 while s ≥ 0 do
   sum := sum + sarr[p];
   s := s - 1;
done
```

The reason we reject this program is that the number of loop iterations, which in general cannot be determined at compile time, could reveal information about the secret at run-time. In this example, the adversary will observe $O(\mathtt{s})$ memory events and thus can infer `s` itself. Prior work on mitigating timing channels often makes the same restriction for the same reason [1], [4], [22], [7]. Similarly, we can mitigate the restrictiveness of our type system by padding out the number of iterations to a constant value. For example, we could transform the above program to be instead

```
 p := N; sum := 0;
 while p ≥ 0 do
   if p < slen then sum := sum + sarr[p];
    else sdummy := sdummy + sarr[p];
   p := p - 1;
done
```

Here, `N` is some constant and `sdummy` and `sum` are allocated in the same ORAM bank. The loop will always iterate `N` times but will compute the same `sum` assuming `N` $\geq$ `slen`.

We also do not allow loops with low guards to appear in a conditional with a high guard. As above, we may be able to transform a program to make it acceptable. For example, for some S, the program `if s > 0 then while (p > 0) do` S`; done` could be transformed to be `while (p > 0) do if s > 0 then` S`; done` (assuming `s` is not written in S). This ensures once again that we do not leak information about the loop guard.

E. Allocating array elements across ORAM banks

For simplicity, our operational semantics and type system model all elements of the same array as all being stored in the same ORAM bank. However, as discussed at the end of Section II and demonstrated empirically in Section V, performance can be significantly improved by allocating each array element in a separate ORAM bank. Here we sketch extensions to the type system and operational semantics that permit each element of an array to be allocated in an ORAM bank of size 1; i.e., the contents are merely encrypted/decrypted on access with no other special handling.[4]

The changes to the operational semantics are straightforward. First, we change memories M to map variables to either to pairs $(n, l) \in \textbf{Nat} \times \textbf{SecLabel}$ or arrays $m \in \textbf{Arrays}$. Arrays are changed to map indexes $n \in \textbf{Nat}$ to pairs $(n, l) \in \textbf{Nat} \times \textbf{SecLabel}$, thus allowing each array element to be in its own ORAM bank. Rules (E-Arr) and (S-AAsn) are updated in the obvious manner, using the cell's individual label l in the event.

The type system is extended as follows. First, we identify a *symbolic ORAM bank* $o^\top \in \textbf{ORAMbanks}$; an array x of type Array $o^\top$ represents an array whose elements are each stored in an ORAM bank of size 1. In addition, we extend the notion of trace patterns to include events $\textbf{SArr}(x, e)$ which indicate a read or write of array `x` whose type is $o^\top$; the indexing expression e is included in the event. We extend trace equivalence to include the axiom $\textbf{SArr}(x, e) \sim_L \textbf{SArr}(x, e)$; i.e., two accesses to an array in the symbolic ORAM bank are equivalent if they have

[4] Note that this extension has no impact on expressiveness: we always have the option of allocating the whole array in the same bank and degenerating to the existing type system.

the same index expression. We modify both the (T-Arr) and (T-AAsn) rules to generate event $\mathbf{SArr}(x, e)$ when l is $o^\top$.

We also extend these two rules to require that when l is $o^\top$ then the label of the index expression e must be L; i.e., we only permit indexing arrays in the symbolic ORAM bank with public values. If we allowed secret indexes, then the adversary could learn something about the index by observing which ORAM bank is read. For example, suppose we had the program `h[x] := y` where `h` has type Array $o^\top$, and `x` and `y` have type Nat o. Suppose the first element of `h` is allocated in ORAM bank o_1 and the second element is in ORAM bank o_2. Then if we run the program when `x` is 0 we get trace $o@o@o_1$ (corresponding to the read of `x`, the read of `y`, and the write of `h[0]`). But if we run the program with `x` is 1, we get trace $o@o@o_2$. Assuming the adversary knows the ORAM allocation for `h` (i.e., assuming he knows Γ) then he has learned something about the value of `x`. On the other hand, allocating all of an array in the same bank eliminates this channel of information, so it is safe to index it with a secret value.

Looking at the code in Figure 1(b), we can see that `h` could be allocated in $o^\top$. This is because `h` is accessed with the same index expression (`i`) on lines 6 and 7, and trace equivalence (which precludes writes to public variables) ensures that `i` will have the same value in both cases. On the other hand, if line 7 was instead `else mdummy = h[i+5]`, then the program would be rejected because the index expressions in both branches are not the same.

Note that for soundness we need to prove that all possible run-time values for an array index are the same on both branches of a secret conditional; preventing writes to public variables in such branches and requiring the index expression to be the same is a simple way to do this. Of course, more sophisticated decision procedures could also be used (that would, for example, know that `i+5 = 5+i`).

IV. Compilation

Rather than requiring programmers to write memory-trace oblivious programs directly, we would prefer that programmers could write arbitrary programs and rely on a compiler to transform those programs to be memory trace oblivious. While more fully realizing this goal remains future work, we have developed a compiler algorithm that automates some of the necessary tasks.

In particular, given a program P in which the inputs and outputs are labeled as `secret` or `public`, our compiler will (a) infer the least labels (`secret` or `public`) for the remaining, unannotated variables; (b) allocate all `secret` variables to distinct ORAM banks; (c) insert padding instructions in conditionals to ensure their traces are equivalent; and finally, (d) allocate instructions appearing in high conditionals to ORAM banks. These steps are sufficient to transform the `max` program in Figure 1(a) into the memory-trace oblivious version in Figure 1(b). We can also transform other interesting algorithms, such as k-means, Dijkstra's shortest paths, and matrix multiplication, as we discuss in the next section.

We now sketch the different steps of our algorithm.

A. Type checking source programs

The first step is to perform *label inference* on the source program to make sure that we can compile it. This is the standard, constraint-based approach to local type inference as implemented in languages like Jif [25] and FlowCaml [32]. We introduce fresh constraint variables for the labels of unannotated program variables, and then generate constraints based on the structure of the program text. This is done by applying a variant of the type rules in Figure 6, having three differences. First, we treat labels l has being either L, representing `public` variables; H, representing `secret` variables (we can think of this as the only available ORAM bank); or α, representing constraint variables. Second, premises like $l_1 \sqsubseteq l_2$ and $l_0 \sqcup l_1 \sqsubseteq l_2$ that appear in the rules are interpreted as *generating* constraints that are to be solved later. Third, all parts having to do with trace patterns T are ignored. Most importantly, we ignore the requirement that $T_1 \sim_L T_2$ for conditionals.

Given a set of constraints generated by an application of these rules, we attempt to find the least solution to the variables α that appear in these constraints, using standard techniques [13]. If we can find a solution, the compilation may continue. If we cannot find a solution, then we have no easy way to make the program memory-trace oblivious, and so the program is rejected.

As an example, consider the `max` program in Figure 1(a), but assume that on lines 3 and 4 the variables `i` and `m` are not annotated, i.e., they are missing the `secret` and `public` qualifiers. When type inference begins, we assign `i` the constraint variable α_i and `m` the constraint variable α_m. In applying the variant type rules (with the PC label l_0 set to L) to this program (that is, the part from lines 5–7), we will generate the following constraints:

$(\alpha_i \sqcup L) \sqcup L \sqsubseteq L$	$line$ 5
$\alpha_i \sqsubseteq H$	$line$ 6, for $h[i]$ in guard
$l_0 = \alpha_i \sqcup H \sqcup \alpha_m \sqcup L$	PC label for checking if branch
$\alpha_i \sqsubseteq H$	$line$ 6, for $h[i]$ in assignment
$l_0 \sqcup (H \sqcup \alpha_i) \sqsubseteq \alpha_m$	$line$ 6, assignment
$L \sqcup (\alpha_i \sqcup L) \sqsubseteq \alpha_i$	$line$ 7

(For simplicity we have elided the constraints on location labels that arise due to (T-Lab), but normally these would be included as well.) We can see that the only possible solution to these constraints is for α_i to be L and α_m to be H, i.e., the former is `public` and the latter is `secret`.

Assuming that the programmer minimally labels the source program, only indicating those data that *must* be secret and leaving all other variables unlabeled, then the main restriction on source programs is the restriction on the use of loops: all loop guards must be public, and no loop may appear in a conditional whose guard is high. As mentioned in the previous section, the programmer may transform such programs into equivalent ones, e.g., by using a constant loop bound, or by hoisting loops out of conditionals. We leave the automation of such transformations to future work.

B. Allocating variables to ORAM banks

Given all variables that were identified as `secret` in the previous stage, we need to allocate them to one or more ORAM banks. At one extreme, we could put all secret variables in a single ORAM bank. The drawback is that each access to a secret variable could cause significant overhead, since ORAM accesses are polylogarithmic in the size of the ORAM [17] (on top of the encryption/decryption cost). At the other extreme, we could put every secret variable in a separate ORAM bank. This lowers overhead by making each access cheaper but will force the next stage to insert more padding instructions, adding more accesses overall. Finally, we could attempt to choose some middle ground between these extreme methods: put some variables in one ORAM bank, and some variables in others.

Ultimately, there is no analytic method for resolving this tradeoff, as the "break even" point for choosing padding over increased bank size, or vice versa, depends on the implementation. A profile-guided approach to optimizing might be the best approach. With our limited experience so far we observe that storing each secret variable in a separate ORAM bank generally achieves very good performance. This is because when conditional branches have few instructions, the additional padding adds only a small amount of overhead compared to the asymptotic slowdown of increased bank size. Therefore we adopt this method in our experiments. Nevertheless, more work is needed to find the best tradeoff in a practical setting.

We also need to assign secret statements (i.e., those statements whose location label must be H) to ORAM banks. At this stage, we assign all statements under a given conditional to the same ORAM bank, but we make a more fine-grained allocation after the next stage, discussed below.

C. Inserting padding instructions

The next step is to insert padding instructions into conditionals, to ensure the final premise of (T-Cond) is satisfied, so that both branches will generate the same traces.

To do this, we can apply algorithms that solve the *shortest common supersequence* problem [14] when applied to two traces (a.k.a. the 2-scs problem). That is, given the two trace patterns T_t and T_f for the true and false branches of an `if` (following ORAM bank assignment), let T_{tf} denote the 2-scs of T_t and T_f. The differences between T_{tf} and the original traces signal where, and what, padding instructions must be inserted. The standard algorithm builds on the dynamic programming solution to the *greatest common subsequence* (gcs) algorithm, which runs in time $O(nm)$ where n and m are the respective lengths of the two traces [8]. Using this algorithm to find the gcs reveals which characters must be inserted into the original strings, as illustrated in Figure 8.

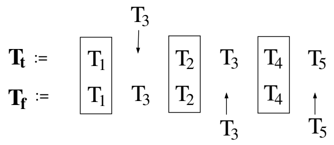

Figure 8. **Finding a short padding sequence using the greatest common subsequence algorithm.** An example with two abstract traces $\mathbf{T}_t = [T_1; T_2; T_3; T_4; T_5]$ and $\mathbf{T}_f = [T_1; T_3; T_2; T_4]$. One greatest common subsequence as shown is $[T_1; T_2; T_4]$. A shortest common supersequence of the two traces is $\mathbf{T}_{tf} = [T_1; T_3; T_2; T_3; T_4; T_5]$.

When running 2-scs on traces, we view T_t and T_f as strings of characters which are themselves trace patterns due to single statements. Each statement-level pattern will always begin with a **Fetch** p, and be followed by zero or more of the following events: **Read**, o_i for ORAM bank i, and in the extended type system, $\textbf{SArr}(x, e)$.[5] For example, suppose we have the program o:**skip**; $o{:}x[y] := z$ where, after ORAM bank assignment, the type of y is Nat o_1, the type of z is Array o_1, and the type of x is Nat o_2. This program generates trace pattern **Fetch** o@**Fetch** $o@o_1@o_1@o_2$. For the purposes of running 2-scs, this trace consists of two characters: (**Fetch** o), which corresponds to the statement o:**skip**, and (**Fetch** $o@o_1@o_1@o_2$), which corresponds to the statement $o{:}x[y] := z$.

Once we have computed the 2-scs and identified the padding characters needed for each trace, we must generate "dummy" statements to insert in the program that generate the same events. This is straightforward. In essence, we can allocate a "dummy" variable d_o for each ORAM bank o in the program, and then read, write, and compute on that variable as needed to generate the correct event. Suppose we had the program $\textbf{if}(e, o{:}\textbf{skip}, o{:}x[y] := z)$ and thus $T_t = \textbf{Fetch}\ o$ and $T_f = \textbf{Fetch}\ o@o_1@o_1@o_2$. Computing the 2-scs we find that T_t can be pre-padded with **Fetch** $o@o_1@o_1@o_2$ while T_f can be post-padded with **Fetch** o. We can readily generate statements that correspond to both. For the second, we produce o:**skip**. For the first, we can produce $o{:}d_{o_2} := d_{o_1} + d_{o_1}$. When we must produce an event corresponding to a public read, or read from an array, we can essentially just insert a read from that variable directly. Finally, for the extended type system, we can simply use the actual expression $x[e]$ to produce an event $\textbf{SArr}(x, e)$,

Note that this approach will generate more padding instructions than is strictly needed. In the above example, the final program will be $\textbf{if}(e, (o{:}d_{o_2} := d_{o_1} + d_{o_1}; o{:}\textbf{skip}), (o{:}x[y] := z; o{:}\textbf{skip}))$. Peephole optimizations

[5] Because of the restrictions imposed by the type system, T_t and T_f will never contain loop patterns, (public) read-array or write patterns, or or-patterns.

Table II
Programs and parameters used in our simulation.

No.	Description
1	Dijkstra ($n = 100$ nodes)
2	K-means ($n = 100$ data points, $k = 2$, $I = 1$ iteraion)
3	Matrix multiplication ($n \times n$ matrix where $n = 40$)
4	Matrix multiplication ($n \times n$ matrix where $n = 25$)
5	Find max ($n = 100$ elements in the array)
6	Find max ($n = 10000$ elements in the array)

can be used to eliminate some superfluous instructions. However, a better approach is to use a finer-grained alphabet which in practice is available when using three address code, i.e., as the intermediate representation of an actual compiler. In this kind of language, which involves adversary-invisible reads/writes to registers, the alphabet can be more fine grained. We have formalized compilation in this style, and give several examples, in our technical report [28].

Once padding has been inserted, both branches have the same number of statements, and thus we can allocate each pair of statements in its own ORAM bank. Assuming we did not drop the **skip** statements in the program above, we could allocate them both in ORAM bank o_3 and allocate the two assignments in ORAM bank o_4, rather than allocate all instructions in ORAM bank o as is the case now.

V. Evaluation

To demonstrate the efficiency gains achieved by our compiler in comparison with the straightforward approach of placing all secret variables in the same ORAM bank, we choose four example programs: Dijkstra single-source shortest paths, K-means, Matrix multiplication (naïve $O(n^3)$ implementation), and Find-max (Figure 1).

We will compare three different strategies:

Strawman: Place all secret variables in the same ORAM bank, and place all code in the same ORAM bank (different from the one for storing data).

Opt 1: Store each variable in a separate ORAM bank, but store whole arrays in the same bank. Allocate instructions to ORAM banks using the algorithm described in Section IV.

Opt 2: Store each variable, and each member of an array, in a separate ORAM bank (when allowed, as per Section III-E). Allocate instructions to ORAM banks using the algorithm described in Section IV.

In all three cases, we insert padding as necessary to ensure obliviousness.

A. Asymptotic Analysis

For the four programs we choose, Table I shows the total number of memory accesses in terms of the data size n. In this table, we assume that such that each access to an ORAM bank of size m requires poly $\log m$ physical memory accesses [17], [27], [36]. The degree of the polylog and the constant would depend on the specific ORAM implementation and the parameter choices.

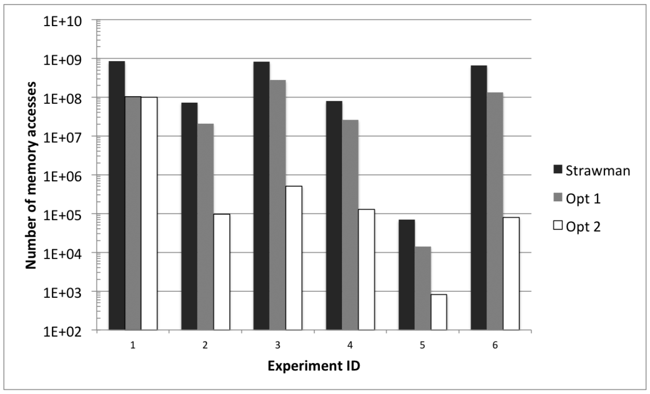

Figure 9. Simulation Results for Strawman, Opt 1, and Opt 2.

Memory accesses due to data. Table I illustrates that Opt 1 does not achieve asymptotic improvements compared to the Strawman (though performance is improved by constant factors, as discussed below). On the other hand, Opt 2 does achieve asymptotic performance gains. The table shows that Opt 2 is a poly $\log(n)$ factor faster than the strawman scheme, particularly for the K-means, Matrix multiplication, and Find-max examples. Intuitively, this is because these algorithms, like a large class of data mining algorithms, traverse the data in a predictable pattern independent of the data contents. Thus it suffices to just encrypt the data, i.e., put each array element in an ORAM bank of size 1.

Memory accesses due to instruction fetches. Table I shows that both Opt 1 and Opt 2 achieve asymptotic savings in terms of number of memory accesses due to instruction fetches. Our compiler avoids placing all code in the same ORAM bank; instead, it stores only instructions on both branches of a conditional with a high guard in ORAM banks, and partitions instructions into smaller ORAM banks whenever possible.

B. Simulation Results

We also performed simulation to measure the performance of the example programs when compiled by our compiler. Code for the source and transformed programs is given in Appendix II. Table II shows the parameters we choose for our experiment. We built a simulator in C++ that can measure the number of memory accesses for transformed programs. Implementing a full-fledged compiler that integrates with our ORAM-capable hardware concurrently being built [29] is left as future work.

Simulation results are given in Figure 9 for the six setups described in Table II. The ORAM scheme we used in the simulation is due to Shi et al [36]. The figure shows that Opt 1 is 1.3 to 5 times faster the strawman scheme; and Opt 2 is *1 to 3 orders of magnitude* faster than the strawman for the chosen programs and parameters.

Table I

Number of memory accesses. c is a small constant (typically 2 or 3) depending on the ORAM scheme chosen. n stands for the size of the data. P is the length of the program. For K-means, the data is 2-dimensional, n is the number of data points, k is the number of clusters, and I is the number of iterations (fixed ahead of time independent of the data).

	Memory accesses for data			Memory accesses for instructions		
Program	Strawman	Opt 1	Opt 2	Strawman	Opt 1	Opt 2
Dijkstra	$O(n^2 \log^c n)$	$O(n^2 \log^c n)$	$O(n^2 \log^c n)$	$O(n^2 P \log^c P)$	$O(n^2 P)$	$O(n^2 P)$
K-means	$O(Ink \log^c n)$	$O(Ink \log^c n)$	$O(Ink)$	$O(InkP \log^c P)$	$O(InkP)$	$O(InkP)$
Matrix multiplication	$O(n^3 \log^c n)$	$O(n^3 \log^c n)$	$O(n^3)$	$O(n^3 P \log^c P)$	$O(n^3 P)$	$O(n^3 P)$
Find max	$O(n \log^c n)$	$O(n \log^c n)$	$O(n)$	$O(nP \log^c P)$	$O(nP)$	$O(nP)$

VI. Related Work

We are the first to approach the problem of achieving privacy using ORAM from a programming languages theory perspective. Previously, the research community has largely considered *generic oblivious program simulation* (i.e., placing all variables in a single ORAM) [17], [12]), which is relatively inefficient and can leak information through the memory trace. Work that has addressed the memory trace channel has not done so formally, and therefore provides no rigorous guarantees [50]. Several others have proposed custom data-oblivious algorithms [20], [11] that achieve asymptotically better performance than generic oblivious simulation, but do not scale in terms of human effort due to the need to design for each specific problem. In comparison, our proposed approach provides a rigorous security guarantee (memory trace obliviousness) and compiles programs so that they achieve this guarantee. By partitioning variables and array contents among multiple ORAM banks we can sometimes asymptotic performance improvements compared to generic simulation.

Oblivious RAM (ORAM) was first proposed by Goldreich and Ostrovsky [17]. Since its proposal, various improvements have been proposed [19], [27], [46], [18], [16], [45]. Our programming language techniques rely on ORAM as a black box; it does not matter which underlying ORAM scheme is employed. Active DRAM capable of handling programmable logic (e.g., the emerging Hybrid Memory Cube technology [6]) can also be used in place of ORAM. In this case, encrypted memory addresses can be transmitted over the bus, and Active DRAM would be able to decrypt those addresses. Our techniques would also readily apply when the underlying hardware realization is Active DRAM.

Our work draws ideas from existing type systems that enforce information flow security [34]. The main difference in our setting is the adversary model: we assume the adversary can view the memory trace, which includes the number and content of events, and program termination. In general, we are more restrictive than type systems that enforce standard, termination-insensitive noninterference, as illustrated with examples in Section III-D. We are the first to state the memory trace obliviousness property, and to present a type system and compiler for enforcing it.

Our requirement that loops have low guards and that conditionals produce equivalent traces is reminiscent of work that transforms programs to eliminate timing channels [1], [4], [22], [7] where inserted padding aims to equalize execution times. Memory trace obliviousness is orthogonal to timing-sensitive noninterference; while methods for enforcing them are similar, programs satisfying the first need not satisfy the second, and vice versa.

VII. Conclusions and future work

This paper has proposed the property of *memory-trace oblivious execution* as a practical requirement of using the XOM model [44] for cloud computing, in which the cloud provider's CPU is trusted, but the rest of the hardware is subject to physical attacks. We have presented a programming language that stores secret data in *oblivious RAM* (ORAM) and defined a type system for proving that programs written in this language ensure memory trace oblivious execution. We have also presented a simple compiler that allocates secret variables to different ORAM banks and performs some simple program transformations toward ensuring programs are safe. Our approach can achieve asymptotic performance gains for many real-world programs compared to storing all variables in a single ORAM.

At present we are considering three avenues of future work. First, we plan to explore how to compose work on mitigating timing channels with our work. One simple composition would perform black-box, predictive mitigation [3] on the output from our compiler (suitably adjusted to ensure we retain our memory obliviousness guarantee). A more language-level integration is also possible [49].

Second, we plan a more architecture-aware development of our ideas. Just as inserted padding can be ineffective for timing channels because it fails to account for chip-level features such as branch predictors and caches [22], these features can confound our attempts ensure oblivious execution traces. For example, instruction and data fetches to main memory might be suppressed because they are present in cache, and this presence may be due to secret information. One solution to this problem is to place such features between the ORAM controller and main memory, but this may be impractical. In the worst case, as Zhang et al., some features can be temporarily disabled (in high contexts) to avoid leaks. We are developing an architectural prototype and simulator to assess various options.

Finally, we plan to develop a more full-featured compiler.

In addition to inserting padding, as happens now, we will explore program transformations for hoisting loops or conditionals, as sketched in Section III-D. We will also account for more language features, such as dynamic memory allocation, which can itself be a source of leaks. We may also incorporate more sophisticated decision procedures for enabling more array elements to be safely stored across ORAM banks.

References

[1] Johan Agat. Transforming out timing leaks. In *27th ACM Symposium On Principles OF Programming Languages (POPL)*, pages 40–53, 2000.

[2] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security*, pages 297–307, 2010.

[4] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, May 2006.

[5] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[6] HMC Consortium. Hybrid memory cube. http://hybridmemorycube.org/.

[7] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 45–60, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms, Third Edition*. MIT Press, 2009.

[9] Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings of the 17th Computer Security Foundations Workshop*, pages 115–124. IEEE, June 2004.

[10] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[11] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[12] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, STC '12, pages 3–8. ACM, 2012.

[13] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 28(6):1035–1087, November 2006.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[15] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association.

[16] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[17] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[18] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 576–587, 2011.

[19] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[20] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[21] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[22] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):163–182, December 2005.

[23] Adam Holt, Keith Weiss, Katy Huberty, Ehud Gelblum, Simon Flannery, Sanjay Devgan, Atif Malik, Nathan Rozof, Adam Wood, Patrick Standaert, Francois Meunier, Jasmine Lu, Grace Chen, Bill Lu, Keon Han, Vipin Khare, and Masaharu Miyachi. Cloud computing takes off, market set to boom as migration accelerates. Morgan Stanley Blue Paper, http://www.morganstanley.com/views/perspectives/cloud_computing.pdf, 2011.

[24] Safety joins performance: Infineon introduces automotive multicore 32-bit microcontroller family aurix-tm to meet safety and powertrain requirements of upcoming vehicle generations. http://www.infineon.com/cms/en/corporate/press/news/releases/2012/INFATV201205-040.html.

[25] Jif: Java + information flow. http://www.cs.cornell.edu/jif/.

[26] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 413–428, 2011.

[27] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[28] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. Technical Report CS-TR-5020, University of Maryland Department of Computer Science, 2013. http://www.cs.umd.edu/~mwh/papers/pl-oram-tr.pdf.

[29] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, John Kubiatowicz, Krste Asanovic, Ch. Papamanthou, and Dawn Song. Practical, on-demand oblivious computation. Manuscript in submission, 2012.

[30] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[31] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.

[32] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[33] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 183–196, 2007.

[34] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[35] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[36] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[37] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2005.

[38] Weidong Shi and Hsien-Hsin S. Lee. Authentication control point and its implications for secure processor design. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, and Chenghuai Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 123–134, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 14–24, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Sergei Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, June 2002.

[42] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[43] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[44] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.*, 34(5):168–177, November 2000.

[45] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[46] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 139–148, New York, NY, USA, 2008. ACM.

[47] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: a parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.

[48] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Commun. ACM*, 54(11):93–101, 2011.

[49] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, pages 99–110, 2012.

[50] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News*, 32(5):72–84, October 2004.

$$\epsilon@t \equiv t@\epsilon \equiv t \qquad \frac{t_1 = t_2}{t_1 \equiv t_2} \qquad \frac{t_1 \equiv t_2}{t_2 \equiv t_1}$$

$$\frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{t_1@t_2 \equiv t_1'@t_2'} \qquad (t_1@t_2)@t_3 \equiv t_1@(t_2@t_3)$$

$$\frac{t_1 \equiv t_2 \quad t_2 \equiv t_3}{t_1 \equiv t_3}$$

Figure 10. Trace equivalence

APPENDIX

I. PROOFS

A. Trace equivalence and lemmas

We shall further study some properties of trace equivalence. First of all, we define the length of a trace t, denoted as $|t|$ to be:

$$|t| = \begin{cases} 1 & \text{if } t = \textbf{read}(x, n) \mid \textbf{write}(x, n) \mid \textbf{fetch}(p) \mid \\ & \quad \textbf{readarr}(x, n, n') \mid \textbf{writearr}(x, n, n') \\ 0 & \text{if } t = \epsilon \\ |t_1| + |t_2| & \text{if } t = t_1@t_2 \end{cases} \tag{1}$$

Lemma 1. *If $t_1 \equiv t_2$, then $|t_1| = |t_2|$.*

Proof: Let us prove by induction on how $t_1 \equiv t_2$ is derived. If $t_1 = t_2$, then the conclusion is obvious. If $t_1 = \epsilon@t_2$, then $|t_1| = |\epsilon| + |t_2| = |t_2|$. Similarly, we can prove the conclusion when $t_1 = t_2@\epsilon$, $t_2 = \epsilon@t_1$, $t_2 = t_1@\epsilon$, or $t_1 = \epsilon@t$ and $t_2 = t@\epsilon$.

If $t_1 = t_{11}@t_{12}$, $t_2 = t_{21}@t_{22}$, $t_{11} \equiv t_{21}$, and $t_{12} \equiv t_{22}$, then by induction, we have $|t_{11}| = |t_{21}|$, and $|t_{12}| = |t_{22}|$. Therefore $|t_1| = |t_{11}| + |t_{12}| = |t_{21}| + |t_{22}| = |t_2|$.

Finally, if $t_1 = (t_1'@t_2')@t_3'$ and $t_2 = t_1'@(t_2'@t_3')$, then $|t_1| = |t_1'@t_2'| + |t_3'| = |t_1'| + |t_2'| + |t_3'| = |t_1'| + |t_2'@t_3'| = |t_2|$. ■

Now, we define the i-th element in a trace, denoted $t[i]$, as follows:

$$t[i] = \begin{cases} \epsilon & \text{if } i \leq 0 \vee i > |t| \\ t & \text{if } i = 1 \wedge t = \textbf{read}(x, n) \mid \textbf{write}(x, n) \mid \\ & \quad \textbf{fetch}(p) \mid \textbf{readarr}(x, n, n') \mid \\ & \quad \textbf{writearr}(x, n, n') \\ t_1[i] & \text{if } t = t_1@t_2 \vee 1 \leq i \leq |t_1| \\ t_2[i - |t_1]\!] & \text{if } t = t_1@t_2 \vee |t_1| < i \leq |t| \end{cases}$$

It is easy to see that if $\forall i.t_1[i] = t_2[i]$ implies $|t_1| = |t_2|$ by the following lemma.

Lemma 2. *$t[i] \neq \epsilon$ for all i such that $1 \leq i \leq |t|$, and ϵ otherwise.*

Proof: The second part of the conclusion is trivial since it directly follows the definition. We prove the first part by induction on $|t|$. If $|t \models 0$, then the conclusion is trivial.

If $|t \models 1$, and $1 \leq i \leq |t|$, then i must be 1. Therefore, $t[i]$ is one of $\textbf{read}(x, n$, $\textbf{write}(x, n)$, $\textbf{fetch}(p)$, $\textbf{readarr}(x, n, n')$, and $\textbf{writearr}(x, n, n')$, and therefore $t[i] \neq \epsilon$.

If $|t| > 1$, then t must be a concatenation of two subsequences, i.e. $t_1@t_2$. If $1 \leq i \leq |t_1|$, then $t[i] = t_1[i]$, and by induction, we know that $t[i] \neq \epsilon$. Otherwise, if $|t_1| < i \leq |t|$, then $0 < i - |t_1| \leq |t \vdash |t_1| = |t_2|$. For natural number n, $n > 0$ implies $n \geq 1$. Therefore $1 \leq i - |t_1| \leq |t_2|$, and by induction, we have $t[i] = t_2[i - |t_1]\!] \neq \epsilon$. ■

Before we go to the next lemma, we shall define the canonical representation of a trace. First, we define the number of blocks in a trace t, denoted by $\#(t)$, as

$$\#(t) = \begin{cases} \#(t_1) + \#(t_2) & \text{if } t = t_1@t_2 \\ 1 & \text{otherwise} \end{cases}$$

Then we define an order $\preceq_t$ between two traces t_1 and t_2 as follows: $t_1 \preceq_t t_2$ if and only if either of the following two conditions hold true: (i) $\#(t_1) < \#(t_2)$, or (ii) $\#(t_1) = \#(t_2) \geq 2$, $t_1 = t_1'@t_1''$, $t_2 = t_2'@t_2''$, and either of the following three sub-conditions holds true: (ii.a) $\#(t_1') > \#(t_2')$; (ii.b) $\#(t_1') = \#(t_2')$ and $t_1' \preceq_t t_2'$; or (ii.c) $t_1' = t_2'$ and $t_1'' \preceq t_2''$. It is easy to see that $\preceq_t$ is complete.

Definition 4 (canonical representation)**.** *The canonical representation of a trace t is the minimal element in the set $\{t' : t \equiv t'\}$ under order $\preceq_t$.*

Lemma 3. *The canonical representation of t is (i) ϵ is $|t \models 0$; or (ii) $can(t) = (...((t_1@t_2)@t_3)...@t_n)$, where $n = |t| > 0$, and $t_i = t[i]$.*

Proof: On the one hand, it is easy to see that $can(t)$ belongs to the set $\{t' : t \equiv t'\}$. In fact, we can prove by induction on $\#(t)$. If $\#(t) = 1$, then either $|t \models 1$, or $|t \models 0$. For the former case, t is one of $\textbf{read}(x, n$, $\textbf{write}(x, n)$, $\textbf{fetch}(p)$, $\textbf{readarr}(x, n, n')$, and $\textbf{writearr}(x, n, n')$, and thus $t = t[1] = can(t)$. For the later case, $t = \epsilon$.

Now suppose $\#(t) > 1$, and thus $t = t'@t''$. Suppose $|t' \models l_1$ and $|t'' \models l_2$. If $l_2 = 0$, by induction, $t'' = \epsilon$, and thus $t \equiv t'$. Furthermore, we have $|t| = |t'|$, and $\forall i.t[i] = t'[i]$ by definition. Therefore $t \equiv t' \equiv can(t') = can(t)$. Similarly, we can prove the conclusion is true when $l_1 = 0$. Now suppose $l_1 > 0$ and $l_2 > 0$, then $can(t') = (...((t_1@t_2)@t_3)...t_{l_1})$, and $can(t'') = (...((t_{l_1+1}@t_{l_1+2})@t_{l_1+3})...@t_{l_1+l_2})$. Then $t \equiv can(t')@can(t'') \equiv can(t)$.

On the other hand, we shall show that $can(t)$ is the minimal one in $\{t' : t \equiv t'\}$. To show this point, we only need to show that for all $t' \equiv can(t)$, we have $can(t) \preceq_t t'$. We prove by induction on n. If $n = 1$, the conclusion is obvious. Suppose $n > 1$ and the conclusion holds true for all $n' < n$.

It is easy to see that $\#(t') > 1$, therefore we suppose $t' = t_l@t_r$. Then we prove that there exists k such that $t_l \equiv (...(t_1@t_2)...@t_k)$ and $t_r \equiv (...(t_{k+1}@t_{k+2})...@t_n)$.

We prove by induction on n and how many steps of equivalent-transitive rule, i.e., $t_1 \equiv t_2 \wedge t_2 \equiv t_3 \Rightarrow t_1 \wedge t_3$, should be applied to derive $can(t) \equiv t'$. If we should apply 0 step, then we know one of the following situations holds: (i) $t' = t''@t_n$ where $t'' \equiv (...(t_1@t_2)...@t_{n-1})$; (ii) $t' = (...(t_1@t_2)...t_{n-2})@(t_{n-1}@t_n)$; (iii) $t' = t@\epsilon$; or (iv) $t' = \epsilon@t$. In any case, our conclusion holds true. Now suppose we need to apply $n > 0$ steps to derive t', where in the $n-1$ step, we derive that $can(t) \equiv t''$ and we can derive $t'' \equiv t'$ without applying the equivalent-transitive rule. Therefore by induction, we know that $t'' = t''_l@t''_r$, and there is k such that $t''_l \equiv (...(t_1@t_2)...@t_k)$ and $t''_r \equiv (...(t_{k+1}@t_{k+2})...@t_n)$. Since we can derive $t'' \equiv t'$ without applying equivalent-transitive rule, we know that one of the following situations holds:

1 $t''_l \equiv t_l$ and $t''_r \equiv t_r$;
2 $t' = \epsilon@t''$;
3 $t' = t''@\epsilon$;
4 $t'' = t'@\epsilon$;
5 $t'' = \epsilon@t'$;
6 $\epsilon@t' = t''@\epsilon$;
7 $t'@\epsilon = \epsilon@t''$;
8 $t''_l = (t_{l1}@t_{l2})$, $t_l \equiv t_{l1}$, and $t_r \equiv t_{l2}@t''_r$ (in this case, we have $(t_{l1}@t_{l2})@t''_r \equiv t_{l1}@(t_{l2}@t''_r))$; and
9 $t''_r = (t_{r1}@t_{r2})$, $t_l \equiv t''_l@t_{r1}$, and $t_r \equiv t_{r2}$ (in this case, we have $t''_l@(t_{r1}@t_{r2}) \equiv (t''_l@t_{r1})@t_{r2}$).

For the first 7 cases, the conclusion is trivial. For Case 8, by induction, we know there are some k' such that $t_{l1} \equiv (...(t_1@t_2)...@t_{k'})$ and $t_{l2} \equiv (...(t_{k'+1}@t_{k'+2})...@t_k)$. Therefore $t_l \equiv (...(t_1@t_2)...@t_{k'})$, and $t_r \equiv (...(t_{k'+1}@t_{k'+2})...@t_k)@(...(t_{k+1}@t_{k+2})...@t_n)$ while the later is equivalent to $(...(t_{k'+1}@t_{k'+2})...@t_n)$. Similarly, we can prove under case 9, the conclusion is also true.

Next, we prove that $can(t) \preceq_t t_l@t_r$. To show this point, by induction, we know $(...(t_1@t_2)...@t_k) \preceq t_l$ and $(...(t_{k+1}@t_{k+2})...@t_n) \preceq t_r$. If either $\#(t_l) > k$ or $\#(t_r) > n-k$, we have $\#(t') > n$ and thus $can(t) \preceq t'$. Suppose $\#(t_l) = k$ and $\#(t_r) = n-k$. If $k < n-1$, then by the definition of $\preceq$, we have $can(t) \preceq_t t'$. Next suppose $k = n-1$, then by induction, we know $(...(t_1@t_2)...@t_{n-1}) \preceq_t t_l$, and thus $can(t) \preceq_t t_l@t_r = t'$. ■

Next is the most important lemma about trace-equivalence.

Lemma 4. *$t_1 \equiv t_2$, if and only if $\forall i.t_1[i] = t_2[i]$.*

Proof: "⇒" Suppose $\forall i.t_1[i] = t_2[i]$. Then by Lemma 4, we know $can(t_1) = can(t_2)$, and thus $t_1 \equiv can(t_1) \equiv can(t_2) \equiv t_2$.

"⇐" Suppose $t_1 \equiv t_2$, then by Lemma 4, we have $can(t_1) \equiv can(t_2)$. Due to both $can(t_1)$ and $can(t_2)$ have the same form, we know they are identical. Therefore, we can conclude that $\forall i.t_1[i] = t_2[i]$. ■

B. Lemmas on trace pattern equivalence

Trace pattern equivalence has similar properties as trace equivalence. If fact, we define the length of a trace pattern T, denoted as $|T|$, to be

$$|T| = \begin{cases} 1 & \text{if } \ T = \textbf{Read}(x) \mid \textbf{Fetch}(p) \\ 0 & \text{if } T = \epsilon \\ |T_1| + |T_2| & \text{if } T = T_1@T_2 \end{cases}$$

Similar to trace, we define the i-th element in a trace pattern T, denoted $T[i]$, as follows:

$$T[i] = \begin{cases} \epsilon & \text{if } i \leq 0 \vee i > |T| \\ T & \text{if } i = 1 \wedge T = \textbf{Read}(x) \mid \textbf{Fetch}(p) \\ T_1[i] & \text{if } T = T_1@T_2 \vee 1 \leq i \leq |T_1| \\ T_2[i - |T_1\rrbracket & \text{if } T = T_1@T_2 \vee |T_1| < i \leq |T| \end{cases}$$

Using exactly the same technique, we can prove the following lemma:

Lemma 5. *$T_1 \sim_L T_2$, if and only if $\forall i.T_1[i] = T_2[i]$.*

To avoid verbosity, we do not provide the full proof here. It is quite similar to the proof of Lemma 4

C. Proof of memory trace obliviousness

To prove Theorem 1, *memory trace obliviousness by typing*, we shall first prove the following lemma:

Lemma 6. *If $\Gamma \vdash e : Nat\ L; T$, then for any two Γ-valid low-equivalent memories M_1, M_2, if $\langle M_1, e\rangle \Downarrow_{t_1} n_1$, $\langle M_2, e\rangle \Downarrow_{t_2} n_2$, then $t_1 = t_2$ and $n_1 = n_2$*

Proof: We use structural induction on expression e to prove this lemma. If e is in form of x, then $\Gamma(x) =$ Nat L, and thus $M_1(x) = M_2(x) = n$ according to the definition of low-equivalence and Γ-validity. Therefore $t_1 = \textbf{read}(x, n) = t_2$, and $n_1 = n_2 = n$.

If e is in form of $e_1\ op\ e_2$, then $\Gamma \vdash e_1 :$ Nat L and $\Gamma \vdash e_2 :$ Nat L. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n'_{ij}$, for $i = 1, 2$, $j = 1, 2$. Then $t_{1j} = t_{2j}$ and $n_{1j} = n_{2j}$ for $j = 1, 2$. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$, and $n_1 = n_{11}\ op\ n_{12} = n_{21}\ op\ n_{22} = n_2$.

Next, we consider the expression in form of $x[e]$. We know that $\Gamma(x) =$ Array L, which implies $\Gamma \vdash e :$ Nat L. Suppose $\langle M_i, e\rangle \Downarrow_{t'_i} n'_i$, then by induction $t'_1 = t'_2$ and $n'_1 = n'_2$. Furthermore, since $M_1 \sim_L M_2$, we have $\forall i \in \textbf{Nat}.M_1(x)(i) = M_2(x)(i)$. Therefore $t_1 = t'_1@\textbf{readarr}(x, n'_1, M_1(x)(n'_1)) = t'_2@\textbf{readarr}(x, n'_2, M_2(x)(n'_2)) = t_2$, and $n_1 = M_1(x)(n'_1) = M_2(x)(n'_2) = n_2$.

Finally, the conclusion is trivial for constant expression. ■

For convenience, we define lab : **Type** → **SecLabels** as:

$$lab(\tau) = \begin{cases} l & \text{if } \ \tau = \text{Int } l \\ l & \text{if } \tau = \text{Array } l \end{cases}$$

Similar to Lemma 6, we can prove the following lemma:

Lemma 7. *If $\Gamma \vdash e : Nat\ l; T$ and $l \in$ **ORAMBanks**, then for any two Γ-valid low-equivalent memories M_1, M_2, if $\langle M_1, e\rangle \Downarrow_{t_1} n_1$, $\langle M_2, e\rangle \Downarrow_{t_2} n_2$, then $t_1 = t_2$*

Proof: If $l = \mathrm{L}$, then the conclusion is obvious by Lemma 6. We only consider $l \in$ **ORAMBanks**. We use structural induction to prove this lemma. If e is in form of x, then according to the definition of Γ-validity and $evt(\cdot)$, we have $t_1 = lab(\Gamma(x)) = t_2$.

If e is in form of $e_1\ op\ e_2$, then $\Gamma \vdash e_1$: Nat l_1 and $\Gamma \vdash e_2$: Nat l_2. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n'_{ij}$, for $i = 1, 2$, $j = 1, 2$. Then $t_{1j} = t_{2j}$, for $j = 1, 2$ by induction. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$.

Finally, we consider the expression in form of $x[e]$. We know that $\Gamma \vdash e$: Nat l'. Suppose $\langle M_i, e\rangle \Downarrow_{t'_i} n'_i$. If $l' = L$, then $t'_1 = t'_2$ by Lemma 6. Otherwise, $l \in$ **ORAMBanks**, and by induction assumption, we have $t'_1 = t'_2$. Since $l \in$ **ORAMBanks**, we know $l = lab(\Gamma(x))$, and thus $t_1 = t'_1@l = t'_2@l = t_2$. ■

Now we shall study the property of trace pattern equivalence. First of all, we have the following lemma:

Lemma 8. *Suppose s and S are a statement and a labeled statement respectively. If $\Gamma, l_0 \vdash s; T$, $l_0 \in$ **ORAMBanks** and $\langle M, s\rangle \Downarrow_t M'$, or $\Gamma, l_0 \vdash S; T$, $l_0 \in$ **ORAMBanks** and $\langle M, S\rangle \Downarrow_t M'$, then $M \sim_L M'$.*

Proof: We prove by induction on the statement s and labeled statement S. Notice that the statement is impossible to be **while** statement. The conclusion is trivial for the statement **skip**.

If s is $x := e$, then $l_0 \subseteq lab(\Gamma(x))$, and thus $lab(\Gamma(x)) \in$ **ORAMBanks**. Therefore $M' = M[x \mapsto (n, l)]$ for some natural number n and some security label l, which implies $M' \sim_L M$. Similarly, if s is $x[e_1] := e_2$, then $lab(\Gamma(x)) \in$ **ORAMBanks**. Furthermore, $\langle M, x[e_1] := e_2\rangle \Downarrow_t M[x \mapsto (m, l)]$ for some mapping m, and some security label $l \in$ **ORAMBanks**. Therefore $M' = M[x \mapsto (m, l)]$, which implies for x such that $M(x) = (n, \mathrm{L})$, we know that $M'(x) = (n, \mathrm{L})$. Therefore $M' \sim_L M$.

Next, let us consider statement **if**(e, S_1, S_2). Then we know either of the two conditions holds true: (1) $\langle M, S_1\rangle \to M'$, and (2) $\langle M, S_2\rangle \to M'$. Since $\Gamma, l_0 \vdash$ **if**$(e, S_1, S_2); T$, we have $\Gamma, l' \vdash S_1; T_1$, and $\Gamma, l' \vdash S_2; T_2$, where $l_0 \sqsubseteq l'$. Therefore we know for either condition, we have $M \sim_L M'$.

Then, for labeled statement $p : s'$, we know by induction that $\Gamma, l_0 \vdash s'; T$, $l_0 \in$ **ORAMBanks**, and $\langle M, s'\rangle \Downarrow_{t'} M'$. Therefore $M \sim_L M'$.

Finally, for sequence of two statements $S_1; S_2$, suppose $\langle M, S_1\rangle \Downarrow_t M_1$, and $\langle M_1, S_2\rangle \Downarrow_{t'} M'$. Then $M \sim_L M_1 \sim_L M'$. ■

According to definition of the trace pattern equivalence, it is obvious to see that, if $T \sim_L T'$, then T is a sequence, whose element each is in the form of **Fetch**(p), **Read**(x), ϵ, and o.

We shall define *a trace t belongs to a trace pattern T, under a memory M, denoted by $t \in T[M]$* as follows:

$$\epsilon \in \epsilon[M] \qquad o \in o[M] \qquad \textbf{fetch}(p) \in \textbf{Fetch}(p)[M]$$

$$\frac{t_1 \in T_1[M] \quad t_2 \in T_2[M]}{t_1@t_2 \in T_1@T_2[M]} \qquad \frac{t \in T[M] \quad T \sim_L T'}{t \in T'[M]}$$

$$\frac{M(x) = (n, \mathrm{L}), n \in \textbf{Nat}}{\textbf{read}(x, n) \in \textbf{Read}(x)[M]}$$

Now, we prove the most important lemma for $t \in T[M]$:

Lemma 9. *$t \in T[M]$ if and only if $|t| = |T|$ and $\forall i.t[i] \in (T[i])[M]$.*

Proof: "$\Rightarrow$" Suppose $|t| = |T|$ and $\forall i.t[i] \in (T[i])[M]$. We prove by induction on $\#(t)$. If $\#(t) = 1$, then the conclusion is trivial. Assume the conclusion holds for all $\#(t') < n$, now suppose $\#(t) = n > 1$. Then we know $t = t_1@t_2$. If $t_1 = \epsilon$, then we know $|t_2| = |t| = |T|$ and $\forall i.t_2[i] = t[i] \in (T[i])[M]$, by induction, we know $t_2 \in T[M]$. Furthermore, we have $t_1 = \epsilon \in \epsilon[M]$, therefore $t_1@t_2 \in \epsilon@T[M]$. Since $\epsilon@T \sim_L T$, we have $t = t_1@t_2 \in T[M]$. A similar arguement shows that if $t_2 = \epsilon$, then we also have $t \in T[M]$.

Now let us consider when $|t_1 \models 0$. By induction, we have $t_1 \in \epsilon[M]$ and $t_2 \in T[M]$, and then again, we have $t \in T[M]$. Similarly, if $|t_2 \models 0$, we also have $t \in T[M]$.

Now assume $|t_1| > 0$ and $|t_2| > 0$, and suppose $T_1 = (...(T_1@T_2)...@T_{|t_1|})$ and $T_2 = (...(T_{|t_1|+1}@T_{|t_1|+2})...@T_{|T|})$. Then by induction, we know that $t_1 \in T_1[M]$ and $t_2 \in T_2[M]$, and thus $t_1@t_2 \in T_1@T_2[M]$. According to Lemma 5, we have $T_1@T_2 \sim_L T$, and thus $t = t_1@t_2 \in T[M]$.

"$\Leftarrow$" We prove by induction on how many steps to derive $t \in T[M]$. Suppose we need only 1 step, then one of the following four conditions is true: (i) $t = \epsilon = T$; (ii) $t = o = T$; (iii) $t =$ **fetch**(p), $T =$ **Fetch**(p); (iv) $t =$ **read**(x, n), $T =$ **Read**(x) and $M(x) = n$. In either case, the conclusion is trivial.

Then suppose we need n step, and the last step is derived from $t = t_1@t_2$, $T = T_1@T_2$, and $t_1 \in T_2[M]$ and $t_2 \in T_2[M]$. Then by induction we have $|t_1| = |T_1|$, $|t_2| = |T_2|$, $\forall i.t_1[i] \in (T_1[i])[M]$, and $\forall i.t_2[i] \in (T_2[i])[M]$. For $i < 1$ or $i > |T|$, then $t[i] = \epsilon = T[i]$, and thus $t[i] \in (T[i])[M]$. If $1 \leq i \leq |T_1|$, then $t[i] = t_1[i]$ and $T[i] = T_1[i]$, and by induction, we have $t[i] \in (T[i])[M]$; if $|T_1| < i \leq |T|$, then $t[i] = t_2[i - |t_1\rrbracket$ and $T[i] = T_2[i - |T_1\rrbracket$, and by induction, we have $t[i] \in (T[i])[M]$.

Finally, suppose we need n step, and the last step is derived from $t \in T'[M]$ and $T' \sim_L T$. Then according to Lemma 5, we know that $\forall i.T'[i] = T[i]$, which also implies that $|T'| = |T|$. By induction, we have $|t| = |T'|$ and $\forall i.t[i] \in (T'[i])[M]$, and therefore, we have $\forall i.t[i] \in (T[i])[M]$ and $|t| = |T|$. ■

We have the following corollaries.

Corollary 1. *If $M_1 \sim_L M_2$, and $t \in T[M_1]$, then $t \in T[M_2]$.*

Proof: By Lemma 9, we only need to show that $\forall i.t[i] \in (T[i])[M_2]$.

Let us prove by structural induction on how $t \in T[M]$ is derived. If $t = \epsilon = T$, or $t = o = T$, or $t = \mathbf{fetch}(p)$ and $T = \mathbf{Fetch}(p)$, or $t = t_1 @ t_2$ and $T = T_1 @ T_2$, then the conclusion is trivial. The only condition we need to prove is when $t = \mathbf{read}(x, n)$, and $T = \mathbf{Read}(x)$. If so, since $t \in T[M_1]$, therefore $M_1(x) = (n, \mathrm{L})$. Since $M_1 \sim_L M_2$, we know that $M_2(x) = (n, \mathrm{L})$. Therefore, we have $t = \mathbf{read}(x, n) \in \mathbf{Read}(x)[M_2] = T[M_2]$.

According to the definition of $T[i]$, we know it is in one of the following four forms: ϵ, $\mathbf{Fetch}(p)$, o, or **Read**. If $T[i] = \epsilon$, then we know $i < 1$ or $i > |T| = |t_1|$. Therefore $t[i] = \epsilon$, and thus $t[i] \in (T[i])[M_2]$. If $T[i] = \mathbf{Fetch}(p)$, then we know $t[i] = \mathbf{fetch}(p)$, and if $T[i] = o$, then we know $t[i] = o$. In both situations, we have $t[i] \in (T[i])[M_2]$. Finally, if $T[i] = \mathbf{Read}(x)$, then we know $t[i] = \mathbf{read}(x, n)$ where $n = M_1[x]$. Since $M_1 \sim_L M_2$, we have $M_2[x] = n$, and thus $t[i] \in (T[i])[M_2]$. ■

Corollary 2. *If $t_1 \in T[M]$ and $t_2 \in T[M]$, then $t_1 \equiv t_2$.*

Proof: Assume $t_1 \in T[M]$, and $t_2 \in T[M]$, according to Lemma 9, we have $|t_1| = |T| = |t_2|, \forall i.t_1[i] \in (T[i])[M]$, and $\forall i.t_2[i] \in (T[i])[M]$. According to the definition of $T[i]$, we know it is in one of the following four forms: ϵ, $\mathbf{Fetch}(p)$, o, or **Read**. If $T[i] = \epsilon$, then we know $i < 1$ or $i > |T| = |t_1| = |t_2|$. Therefore $t_1[i] = t_2[i] = \epsilon$. If $T[i] = \mathbf{Fetch}(p)$, then we know $t_1[i] = t_2[i] = \mathbf{fetch}(p)$, and if $T[i] = o$, then we know $t_1[i] = t_2[i] = o$. Finally, if $T[i] = \mathbf{Read}(x)$, then we know $t_1[i] = \mathbf{read}(x, n_1)$, $n_1 = M[x]$, $t_2[i] = \mathbf{read}(x, n_2)$, and $n_2 = M[x]$. Therefore $n_1 = n_2$, and thus $t_1[i] = t_2[i]$. Therefore $\forall i.t_1[i] = t_2[i]$, and according to Lemma 4, we have $t_1 \equiv t_2$. ■

Then we have the following lemmas:

Lemma 10. *Suppose $\Gamma \vdash e : \tau; T$, $T \sim_L T'$ for some T', and memory M is Γ-valid. If $\langle M, e\rangle \Downarrow_t n$, then $t \in T[M]$.*

Proof: We prove by structural induction on e. If e is n, then $T = \epsilon = t$.

If e is x, then $T = evt(lab(\Gamma(x)), \mathbf{Read}(x))$. If $lab(\Gamma(x)) = l \in \mathbf{ORAMBanks}$, then $t = l \in l[M]$. If $lab(\Gamma(x)) = \mathrm{L}$, then $T = \mathbf{Read}(x)$, and $t = \mathbf{read}(x, n)$, where $M(x) = (n, \mathrm{L})$. According to the definition, we know $t \in T[M]$.

If e is $e_1\ op\ e_2$, then suppose $\langle M, e_i\rangle \Downarrow_{t_i} n_i$ and $\Gamma \vdash e_i : l_i; T_i$ for $i = 1, 2$. Then according to the induction assumption, we have $t_i \in T_i[M]$ for $i = 1, 2$. Since $t = t_1 @ t_2$, and $T = T_1 @ T_2$, we know $t \in T[M]$.

Next we consider $x[e']$. Suppose $\Gamma \vdash e' : \mathrm{Nat}\ l'; T'$, and $\langle M, e'\rangle \Downarrow_{t'} n'$, then $T = T' @ evt(lab(\Gamma(x)), \mathbf{Readarr}(x))$, and $t = t' @ evt(lab(\Gamma(x)), \mathbf{readarr}(x, n', n''))$ for some n''. Moreover, we have $t' \in T'[M]$ by induction. Since $T \sim_L T'$, we know $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$. Therefore $t = t' @ lab(\Gamma(x)) \in T' @ lab(\Gamma(x))[M] = T[M]$. ■

Lemma 11. *Suppose $\Gamma, l_0 \vdash S; T$, where S is normal statement or labeled statement, $T \sim_L T'$ for some T', and $l_0 \in$ **ORAMBanks**, and M is a Γ-valid memory. If $\langle M, S\rangle \Downarrow_t M'$, then $t \in T[M]$.*

Proof: We prove by structural induction on the statement S. Since $l_0 \neq \mathrm{L}$, therefore we know S cannot be a **while** statement. If S is **skip**, then $T = \epsilon = t$.

Let us consider when S is $x := e$. Then $\langle M, e\rangle \Downarrow_{t'} n'$, and $\Gamma \vdash e : \tau; T'$, and $T = T' @ evt(lab(\Gamma(x)), \mathbf{Write}(x))$. Since $T \sim_L T$, T does not contain $\mathbf{Write}(x)$, and thus $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$. Therefore $t = t' @ lab(\Gamma(x)) \in T' @ lab(\Gamma(x))[M]$ by Lemma 10.

Next, suppose S is $x[e_1] = e_2$. Suppose $\langle M, e_i\rangle \Downarrow_{t_i} n_i$, and $\Gamma \vdash e_i : \tau; T_i$ for $i = 1, 2$ by induction. Then $t_i \in T_i[M]$ for $i = 1, 2$. Similar to the discussion for $x := e$, we know $lab(\Gamma(x)) \in \mathbf{ORAMBanks}$, and thus $t = t_1 @ t_2 @ lab(\Gamma(x)) \in T_1 @ T_2 @ lab(\Gamma(x))[M]$.

Next, let us consider $(if)(e, S_1, S_2)$. Then $\Gamma, l_0 \vdash S_i; T_i$ for $i = 1, 2$, and $T_1 \sim_L T_2$. As well $\Gamma \vdash e : \tau; T_e$, $\langle M, e\rangle \Downarrow_{t_e} n_e$, $idx = ite(n_e, 1, 2)$, and $\langle M, S_{idx}\rangle \Downarrow_{t_{idx}} M'$. Then $T = T_e @ T_1$, and $t_e \in T_e[M]$. If $idx = 1$, then $\langle M, S_1\rangle_{t_1} M'$, and thus $t_1 \in T_1[M]$. Therefore $t = t_e @ t_1 \in T_e @ T_1[M] = T[M]$. Similarly, if $idx = 2$, then $\langle M, S_2\rangle_{t_2} M'$, and thus $t_2 \in T_2[M]$. Therefore $t_2 \in T_1[M]$. As a conclusion $t = t_e @ t_2 \in T_e @ T_1[M] = T[M]$.

Now consider the labeled statements S. If S is $p{:}s$, then $\Gamma, l_0 \vdash s; T'$, and $\langle M, s\rangle \Downarrow_{t'} M'$. Therefore $T = \mathbf{Fetch}(p) @ T'$, and $t = \mathbf{fetch}(p) @ t'$. By induction assumption, we have $t' \in T'[M]$. Therefore $t \in T[M]$.

Finally, suppose S is $S_1; S_2$. Then we know $\Gamma, l_0 \vdash S_i; T_i$ for $i = 1, 2$, $\langle M, S_1\rangle \Downarrow_{t_1} M'$, and $\langle M', S_2\rangle \Downarrow_{t_2} M''$. Since $l_0 \in \mathbf{ORAMBanks}$, we know $M \sim_L M' \sim_L M''$. By induction assumption, we know $t_1 \in T_1[M]$, and $t_2 \in T_2[M']$. Since $M \sim_L M'$, according to Corollary 1, we know $t_2 \in T_2[M]$. Therefore $t = t_1 @ t_2 \in T_1 @ T_2[M] = T[M]$. ■

Lemma 12. *Suppose $\Gamma, l_0 \vdash S_i; T_i$, for $i = 1, 2$, where $l_0 \in$ **ORAMBanks**, and $T_1 \sim_L T_2$. Given two Γ-valid low-equivalent memories M_1, M_2, if $\langle M_1, S_1\rangle \Downarrow_{t_1} M_1'$, and $\langle M_2, S_2\rangle \Downarrow_{t_2} M_2'$, then $M_1' \sim_L M_2'$, and $t_1 \equiv t_2$.*

Proof: According to Lemma 11, we know that $t_i \in T_i[M_i]$ for $i = 1, 2$. According to Lemma 8, we know that $M_1' \sim_L M_1$ and $M_2' \sim_L M_2$. Since $M_1 \sim_L M_2$, we know that $M_1' \sim_L M_1 \sim_L M_2 \sim_L M_2'$. Because $t_1 \in T_1[M_1]$, and $M_1 \sim_L M_2$, therefore $t_1 \in T_1[M_2]$. Furthermore, since

$T_1 \sim_L T_2$, we have $t_1 \in T_2[M_2]$. Finally, since $t_2 \in T_2[M_2]$, and according to Corollary 2 we have $t_1 \equiv t_2$. ■

Now we are ready to prove Theorem 1.

Proof of Theorem 1: We extend this conclusion by considering both normal statement and labeled statement, and shall prove by induction on the statement s. For notational convention, we suppose $\langle M_1, s\rangle \Downarrow_{t_1} M_1'$, and $\langle M_2, s\rangle \Downarrow_{t_2} M_2'$, and thus $\Gamma, l_0 \vdash S; T$ with $M_1 \sim_L M_2$ and both are Γ-valid. Our goal is prove $t_1 \equiv t_2$, and $M_1 \sim_L M_2$.

If s is **skip**, it is obvious.

Suppose s is $x := e$, then $\Gamma \vdash e : \text{Nat } l; T$. Suppose $\langle M_i, e\rangle \Downarrow_{t_i'} n_i$, for $i = 1, 2$. According to Lemma 6 and Lemma 7, we know $t_1' = t_2'$. If $lab(\Gamma(x)) \in$ **ORAMBanks**, then $M_1' = M_1[x \mapsto (n_1, l_1)] \sim_L M_1 \sim_L M_2 \sim_L M_2[x \mapsto (n_2, l_2)] = M_2'$, and $t_1 = t_1'@lab(\Gamma(x)) = t_2'@lab(\Gamma(x)) = t_2$, which implies $t_1 \equiv t_2$.

If $\Gamma(x) = \text{Nat L}$, then we know $\Gamma \vdash e : \text{Nat L}; T$, and according to Lemma 6, we have $n_1 = n_2$. Then we also have $M_1' = M_1[x \to n_1] \sim_L M_2[x \to n_2] = M_2'$, and $t_1 = t_1'@\textbf{read}(x, n_1) \equiv t_2'@\textbf{read}(x, n_2) = t_2$.

Next, suppose s is $x[e_1] := e_2$. Suppose $\langle M_i, e_j\rangle \Downarrow_{t_{ij}} n_{ij}$ for $i = 1, 2$, $j = 1, 2$. If $lab(\Gamma(x)) = \text{L}$, then we know $\Gamma \vdash e_j : \text{Nat L}$, for $j = 1, 2$. Then by Lemma 6, we have $t_{1j} = t_{2j}$, which implies $t_{1j} \equiv t_{2j}$, $n_{1j} = n_{2j}$, and according to the definition of Γ-validity and low-equivalence, $\forall i. M_1(x)(i) = M_2(x)(i)$. Therefore $t_1 = t_{11}@t_{21}@\textbf{writearr}(x, n_{11}, n_{12}) \equiv t_{21}@t_{22}@\textbf{writearr}(x, n_{21}, n_{22}) = t_2$, and $M_1' = M_1[x \to M_1(x)[n_{11} \to n_{12}]] \sim_L M_2[x \to M_2(x)[n_{21} \to n_{22}]] = M_2'$.

Otherwise, if $\Gamma(x) \in$ **ORAMBanks**, suppose $\Gamma \vdash e_i : \textbf{Nat } l_i; T_i$, for $i = 1, 2$. Then we know $l_0 \sqcup l_1 \sqcup l_2 \sqsubseteq lab(\Gamma(x))$. Therefore, by Lemma 7, based on the same reasoning as above for Nat l case, we have $t_1 = t_{11}@t_{21}@\Gamma(x) \equiv t_{21}@t_{22}@\Gamma(x) = t_2$. Furthermore, $M_1' = M_1[x \to m_1] \sim_L M_1 \sim_L M_2 \sim_L M_2[x \to m_2] = M_2'$ for some two mappings, m_1 and m_2.

Then suppose the statement is **if**(e, S_1, S_2). There are two situations. If $\Gamma \vdash e : \text{Nat } l_e; T_e$, where $l_e \sqcup l_0 \in$ **ORAMBanks**, then according to Lemma 12, we know $M_1' \sim_L M_2'$, and $t_1 \equiv t_2$. Otherwise, we have $l_e = \text{L}$ and $l_0 = \text{L}$. Suppose $\langle M_i, e\rangle \Downarrow_{t_i'} n_i$, for $i = 1, 2$, then according to Lemma 6, we know $t_1' = t_2'$, which implies $t_1' \equiv t_2'$, and $n_1 = n_2$. If $ite(n_1, 1, 2) = 1$, then we know $\langle M_1, S_1\rangle \Downarrow_{t_1''} M_1'$ and $\langle M_2, S_1\rangle \Downarrow_{t_2''} M_2'$. Therefore $t_1 = t_1'@t_1'' \equiv t_1'@t_2'' = t_2$, and $M_1' \sim_L M_2'$ by induction. We can show the conclusion for $ite(n_1, 1, 2) = 2$ similarly.

Next, let us consider the statement **while**(e, S). We know $\Gamma \vdash e : \text{Nat L}; T$, therefore there exists a constant n, and a trace t, such that $\langle M_i, e\rangle \Downarrow_t, n$ for both $i = 1, 2$, by Lemma 6.

We prove by induction on how many steps applying the S-WhileT rule and S-WhileF rule (WHILE rules for short) to derive $\langle \textbf{while}(e, S), M_1\rangle \Downarrow_{t_1}\rangle$. If we only apply one time, then we must apply S-WhileF rule, and thus $n = 0$. Then we have $t_1 = t = t_2$, and $M_1' = M_1 \sim_L M_2 = M_2'$. Suppose the conclusion is true when we need to apply $n - 1$ steps of WHILE rules, now let us consider when we need to apply $n > 0$ steps. Then we know $n \neq 0$. Suppose $\langle M_i, S\rangle \Downarrow_{t_{i1}} M_{i1}$, and $\langle M_{i1}, \textbf{while}(e, S)\rangle \Downarrow_{t_{i2}} M_i'$, for $i = 1, 2$. Then we know that we need to apply $n - 1$ steps of WHILE rules to derive $\langle M_{11}, \textbf{while}(e, S)\rangle \Downarrow_{t_{12}} M_1'$. By induction, we have $t_{11} = t_{21}$, $t_{12} = t_{22}$, and $M_{11} \sim_L M_{21}$. Therefore $M_1' \sim_L M_2'$, and $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$.

Then let us consider label statement $p{:}s'$. Suppose $\langle M_i, s'\rangle \Downarrow_{t_i'} M_i'$. Then $t_1' \equiv t_2'$, and $M_1' \sim_L M_2'$ by induction. Therefore $t_1 = \textbf{fetch}(p)@t_1' = \textbf{fetch}(p)@t_2' = t_2$.

Finally, let us consider $S_1; S_2$. Suppose $\Gamma, l_0 \vdash S_1; T_1$, $\Gamma, l_0 \vdash S_2; T_2$, $\langle M_i, S_1\rangle \Downarrow_{t_{i1}} M_{i1}$, and $\langle M_{i1}, S_2\rangle \Downarrow_{t_{21}} M_i'$. Then by induction assumption, we have $t_{11} = t_{21}$, $t_{12} = t_{22}$, $M_{11} \sim_L M_{12}$, and thus $M_1' \sim_L M_2'$. Therefore $t_1 = t_{11}@t_{12} = t_{21}@t_{22} = t_2$. ■

II. Benchmark programs

Source Program for Dijkstra shortest paths (in C).

```
secret int dijstra(public int n, public int s,
      public int t, secret int e[][MAX]) {
    int vis[MAX];
    int dis[MAX];
    memset(vis, 0, sizeof(vis));
    memset(dis, 0, sizeof(dis));
    vis[s] = 1;
    for(int i=0; i<n; ++i)
        dis[i] = e[s][i];
    for(int i=1; i<n; ++i) {
      int bestj = -1;
      for(int j=0; j<n; ++j)
        if(!vis[j] && (bestj < 0 ||
              dis[j] < dis[bestj]))
           bestj = j;
      vis[bestj] = 1;
      for(int j=0; j<n; ++j)
         if(!vis[j] && (dis[bestj] +
                e[bestj][j] < dis[j]))
             dis[j] = dis[bestj] + e[bestj][j];
    }
    return dis[t];
}
```

Target Program for Dijkstra shortest paths.

```
secret int dijstra(public int n, public int s,
      public int t, secret int e[]) {
    secret int vis[MAX];
    secret int dis[MAX];
    public int i = 0;
    while (i<n) {
        vis[i]=0; dis[i] = 0;
        i = i + 1;
    }
    vis[s] = 1;  i = 0;
    while (i<n) {
       dis[i] = e[s*n+i]; i = i+1;
    }
    i = 1;
    while(i<n) {
        secret int bestj = -1;
        public int j = 0;
        while (j<n) {
           if (!vis[j] && (bestj <0 ||
                   dis[j]<dis[bestj])) {
              bestj = j;
           } else {
              dummybestj = j;
           }
           j=j+1;
        }
        vis[bestj] = 1;
        j=0;
        while (j<n) {
          if(!vis[j] && (dis[bestj] +
               e[bestj*n+j] < dis[j]))
             dis[j]=dis[bestj]+e[bestj*n+j];
          else {
             dummydis[j]=dis[bestj]+e[bestj*n+j];
          }
          j=j+1;
        }
        i=i+1;
    }
    return dis[t];
}
```

Source Program for k-means (in C).

```
void kmeans(public int n, public int k,
    public int T, secret float point[][DIM],
    secret float ans[][DIM])
{
    // suppose the initial guess of ans
    // is already assigned.
    float nans[MAXK][DIM] = {};
    int count[MAXK];
    int t = 0;
    while(t<T) {
        memset(nans, 0, sizeof(nans));
        memset(count, 0, sizeof(count));
        for(int i=0; i<n; ++i) {
            int bestj=0;
            float best = dis(point[i], ans[0]);
            for(int j=1; j<k; ++j) {
                float nb = dis(point[i], ans[j]);
                if(nb < best) {
                    best = nb;
                    bestj = j;
                }
            }
            for(int j=0; j<DIM; ++j)
                nans[bestj][j] += point[i][j];
            count[bestj] ++;
        }
        for(int i=0; i<k; ++i)
            for(int j=0; j<DIM; ++j)
                ans[i][j]/=count[i];
        t=t+1;
    }
}
```

Target Program for k-means.

```
void kmeans(public int n, public int k, public int T
    secret float point[], secret float ans[])
{
  // suppose the initial guess of ans
  // is already assigned.
  secret float nans[MAXK*DIM];
  secret int count[MAXK];
  public int t = 0;
  while(t<T) {
    public int i=0;
    public int j;
    while(i<k) {
      count[i] = 0; j=0;
      while(j<DIM) {
        nans[i*DIM+j]=0;
        j=j+1;
      }
      i=i+1;
    }
    i=0;
    while(i<n) {
      secret int bestj=0;
      secret float best = dis(point+i*DIM, ans);
      j=1;
      while(j<k) {
        secret float nb =
               dis(point+i*DIM, ans+j*DIM);
        if(nb < best) {
           best = nb;
           bestj = j;
        } else {
           dummybest = nb;
           dummybestj = j;
        }
        j=j+1;
      }
      j=0;
      while(j<DIM) {
         nans[bestj*DIM+j]+=point[i*DIM+j];
         j=j+1;
      }
      count[bestj]=count[bestj]+1;
      i=i+1;
    }
    i=0;
    while(i<k) {
       j=0;
       while(j<DIM) {
          ans[i*DIM+j]=nans[i*DIM+j]/count[i];
           j=j+1;
       }
       i=i+1;
    }
    t=t+1;
  }
}
```

Source Program for Matrix Multiplication.

```
void matmul(public int n, secret int a[][MAX],
      secret int b[][MAX], secret int c[][MAX]) {
    for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j) {
            c[i][j] = 0;
            for(int k=0; k<n; ++k)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Target Program for Matrix Multiplication.

```
void matmul(public int n, secret int a[][MAX],
      secret int b[][MAX], secret int c[][MAX]) {
    public int i=0;
    public int j, k;
    while(i<n) {
        j=0;
        while(j<n) {
            c[i*n+j] = 0; k=0;
            while(k<n) {
               c[i][j] += a[i][k] * b[k][j];
               k=k+1;
            }
            j=j+1;
        } i=i+1;
    }
}
```