Memory Trace Oblivious Program Execution

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.

This work is 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 forting 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, 43], 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 [33, 39, 40, 43, 44]. 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 [12, 42, 47], 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 [11, 20]. 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, building 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 do 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 \[^1\] 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 \[\text{III}\] and \[\text{IV}\] present our type system and compiler, and Section \[\text{V}\] presents our empirical evaluation. We compare to related work in Section \[\text{VI}\] and conclude with ideas for future research in Section \[\text{VII}\].

\section{Overview}

Consider a scenario where 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 \[\text{I(a)}\] depicts a motivating example. The program \[\text{max}\] scans through the array \[\text{h[]}\], and finds its maximum element. The length of the array \[\text{n}\] is labeled public, i.e., it may be learned by the adversary. The array \[\text{h[]}\] itself is labeled secret, as is the type of the returned value, i.e., the adversary should know 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 \(\text{h}[]\) or the result. We can see that this program would be accepted by a type system for enforcing standard noninterference \[^{[44]}\].

\subsection{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 assume 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 \[^{[38,39,43,44]}\], 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 \[^{[11,10,26]}\]. 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 \[\text{VI}\] 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 \[\text{VII}\].

\subsection{B. Approach}

We now detail our approach step by step.

\textbf{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

\[^1\]The typing of arrays is slightly non-standard: as explained in the next section, we permit public indexes to secret arrays by using a semantics in the style of Deng and Smith’s “lenient” semantics, which ignores out-of-bounds accesses \[^{[9]}\].
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, 35, 37] 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 [43, 44, 50].

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, 7, 22]. 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 ndummy. 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 ndummy 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 ndummy).

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 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 \( m_{\text{dummy}} \) 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 \( m_{\text{dummy}} \) 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 \( h[i] \). This is because the access pattern on the array does not depend on any secret—even 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 \( h[i] \) in a bank of size 1, and \( m \) or \( m_{\text{dummy}} \) 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 \( m_{\text{dummy}} \), for a total of \( 6n \) accesses.

In comparison, the naive 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.

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 \( \text{skip} \), 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 3, which refers to auxiliary functions and syntax defined in Figure 4. 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, \( (M,S) \downarrow M' \) (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 $|n|$ to model the length of the array; that is, if $|n| = n$ 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 $\text{read}(x, n)$ which states that number $n$ was read from variable $x$ and $\text{read}(x, n_1, n_2)$, 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 $\text{fetch}(p)$ 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 $\text{read}$ 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, $\text{fetch}$ represents the concatenation of two traces and $\epsilon$ 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 $\text{read}$ function defined in Figure 3 if the label is some ORAM bank $o$ then event $o$ will be emitted, otherwise event $\text{read}(x, n)$ 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 $\text{get}$, also defined in Figure 3 that returns 0 if the index $n$ is out of bounds. Rule (S-WhileF) uses the $\text{upd}$ function similarly; if the write is out of bounds, then the array is not affected. 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 and writes with a try-catch block that ignores the exception, which is a common pattern, e.g., in Jif [5] [25], and has also been advocated by Deng and Smith [9].

The rule (S-Cond) for conditionals is the obvious one; we write $\text{ite}(x, y, z)$ 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

\[
\begin{align*}
\langle M, e \rangle \Downarrow_t n \\
E-\text{Var} &\quad t = \text{eval}(l, \text{read}(x, n)) \\
E-\text{Const} &\quad \langle M, n \rangle \Downarrow_e n \\
E-\text{Op} &\quad \langle M, e_1 \rangle \Downarrow_t n_1, \langle M, e_2 \rangle \Downarrow_t n_2, n = n_1 \text{ op } n_2 \\
E-\text{Arr} &\quad \langle M, x[e] \rangle \Downarrow_{e \Downarrow_t} \langle M, x \rangle \\
S-\text{Skip} &\quad \langle M, \text{skip} \rangle \Downarrow_t M' \\
S-\text{Asn} &\quad \langle M, e \rangle \Downarrow_t n, \langle M(x) = (n', l) \rangle \\
S-\text{Asn} &\quad \langle M, x := e \rangle \Downarrow_t M[x := (n, l)] \\
S-\text{Asn} &\quad \langle M, e_1 \rangle \Downarrow_t n_1, \langle M, e_2 \rangle \Downarrow_t n_2, M(x) = (m, l) \\
S-\text{Cond} &\quad \langle M, \text{if} (e, S_1, S_2) \rangle \Downarrow_{t_1 \text{ or } t_2} M' \\
S-\text{WhileT} &\quad \langle M, \text{while}(e, S) \rangle \Downarrow_{\text{fetch}(p) \text{ or } t} M' \\
S-\text{WhileF} &\quad \langle M, \text{while}(e, S) \rangle \Downarrow_0 M' \\
P-\text{Stmt} &\quad \langle M, S \rangle \Downarrow_t M' \\
P-\text{Stmt} &\quad \langle M, p:s \rangle \Downarrow_{t = \text{fetch}(p)} M' \\
P-\text{Stmt} &\quad \langle M, S_1; S_2 \rangle \Downarrow_{t_1 \text{ or } t_2} M'' \\
\end{align*}
\]
(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 \( \forall x, M_1(x) = (v, L) \iff M_2(x) = (v, L) \).

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

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

\[
\Gamma(x) = \text{Nat} l \iff \exists n \in \text{Nat}. M(x) = (n, l) \\
\Gamma(x) = \text{Array} l \iff \exists m \in \text{Arrays}. M(x) = (m, l)
\]

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

**Definition 3** (Memory trace obliviousness). Given a security environment \( \Gamma \), a program \( S \) satisfies \( \Gamma \)-memory trace obliviousness if for any two \( \Gamma \)-valid memories \( M_1 \sim_L M_2 \), if \( (M_1, S) \Downarrow t_1, M'_1 \) and \( (M_2, S) \Downarrow t_2, M'_2 \), then \( t_1 \equiv_L t_2 \), and \( M'_1 \sim_L M'_2 \).

In this definition, we write \( t_1 \equiv_L t_2 \) to denote that \( t_1 \) and \( t_2 \) are equivalent. Equivalence is defined formally in our technical report [23]. Intuitively, two traces are equivalent if they are syntactically equivalent or we can apply associativity to transform one into the other. Furthermore, \( \epsilon \) plays the role of the identity element.

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 \( \Gamma \), expression \( e \) has type \( \tau \), 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 \( \text{Read}(x, n) \in \text{Read}(x) \) 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, \( \text{Loop}(p, T_1, T_2) \), denotes the set of patterns \( \text{Fetch}(p)@T_1 \) and \( \text{Fetch}(p)@T_2 \) which matches traces \( t \) such that either \( t \in T_1 \) or \( t \in T_2 \).
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-Con) 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 $\Gamma \vdash e : \tau; T$ and $\Gamma (x) = \text{Nat } l$ thus looping over arrays reveals no information about them. 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 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.
\[
\begin{align*}
T \sim_L T & \quad \epsilon \sim_L \epsilon \\
\epsilon @ T \sim_L T @ \epsilon & \sim_L T & T_1 \sim_L T_2 & T_2 \sim_L T_3 & T_1 \sim_L T_3 \\
T_1 \circ T_2 \sim_L T_1' \circ T_2' & T_1 \circ T_2 \sim_L T_1' \circ T_2' & (T_1 \circ T_2) @ T_3 & \sim_L T_1 \circ (T_2 @ T_3) \\
o \sim_L o & \quad \text{Fetch } p \sim_L \text{Fetch } p & \quad \text{Read } x \sim_L \text{Read } x
\end{align*}
\]

Figure 7. Trace pattern equivalence

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

The full proof can be found in our technical report [28].

**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

```plaintext
if s > 0 then
p := p; sum := 0;
while s ≥ 0 do
  sum := sum + sarr[p];
  s := s - 1;
done
```

is not accepted in our 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

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

is 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 but rejected in ours:

```plaintext
if s > 0 then
  while (p > 0)
    done
  s := s - 1;
done
```

if \( p < \text{slen} \) then sum := sum + sarr[p];
else sdummy := sdummy + sarr[p];
p := p - 1;
done

Here, \( \text{slen} \) is some constant and \( \text{sdummy} \) and \( \text{sum} \) are allocated in the same ORAM bank. The loop will always iterate \( \text{slen} \) times but will compute the same \( \text{sum} \) assuming \( \text{slen} \geq \text{sdummy} \).

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

```plaintext
if s > 0 then
  while (p > 0)
    done
  s := s - 1;
done
```

do if s > 0 then \( S \); 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.

The changes to the operational semantics are straightforward. First, we change memories \( M \) to map variables to either to pairs \( (n, l) \in \text{Nat} \times \text{SecLabel} \) or arrays \( n \in \text{Arrays} \). Arrays are changed to map indexes \( n \in \text{Nat} \) to pairs \( (n, l) \in \text{Nat} \times \text{SecLabel} \), 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 \( o^T \in \text{ORAMbanks} \); an array \( x \) of type Array \( o^T \) 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

\( \text{SArr}(x, e) \) which indicate a read or write of array \( x \) whose type is \( o^T \); the indexing expression \( e \) is included in the event. We extend trace equivalence to include the axiom

\( \text{SArr}(x, e) \sim_L \text{SArr}(x, e) \); i.e., two accesses to an array in the symbolic ORAM bank are equivalent if they have the same index expression. We modify both the (T-Arr) and (T-AAsn) rules to generate event \( \text{SArr}(x, e) \) when \( l \) is \( o^T \).

We also extend these two rules to require that when \( l \) is \( o^T \) then the label of the index expression \( e \) must be \( L \); i.e., we only permit indexing arrays in the symbolic ORAM bank.

\(^4\text{Note that this extension has no impact on expressiveness: we always have the option of allocating the whole array in the same bank and degenerating to the existing type system.}\)
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 \( o^T \),
and \( x \) and \( y \) have type \( \text{Nat} \ o \). 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 \( o_0 o_0 o_1 \) (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 \( o_0 o_0 o_2 \). Assuming the adversary
knows the ORAM allocation for \( h \) (i.e., assuming he knows \( \Gamma \))
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 include
it with a secret value.

Looking at the code in Figure 1(b), we can see that \( h \)
could be allocated in \( o^T \). 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)
enforces that \( i \) will have the same value in both cases.
On the other hand, if line 7 was instead \( \text{else} \) \( \text{mdummy} := h[i] + 5 \)
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 \text{secret} or \text{public}, our compiler will
(a) infer the least labels (\text{secret} or \text{public}) for the remaining, unannotated variables; (b) allocate all \text{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 \text{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 3 having three differences. First, we treat labels \( l \) has being either \( L \), representing public variables; \( H \), representing \text{secret} variables (we can think of this as the only available ORAM bank); or \( \alpha \), representing constraint variables. Second, premises like \( l_1 \subseteq l_2 \) and \( l_0 \cup l_1 \subseteq l_2 \) 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 \( \alpha \) 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 \text{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 \text{secret} and
\text{public} qualifiers. When type inference begins, we assign \( i \)
the constraint variable \( \alpha_i \) and \( m \) the constraint variable \( \alpha_m \).
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:

\[
\begin{align*}
(\alpha_i \cup L) \cup L & \subseteq L & \text{line 5} \\
\alpha_i & \subseteq H & \text{line 6, for } h[i] \text{ in guard} \\
l_0 = \alpha_i \cup H & \cup \alpha_m \cup L & \text{PC label for checking if branch} \\
\alpha_i & \subseteq H & \text{line 6, for } h[i] \text{ in assignment} \\
l_0 \cup (H \cup \alpha_i) & \subseteq \alpha_m & \text{line 6, assignment} \\
L & \cup (\alpha_i \cup L) & \subseteq \alpha_i & \text{line 7}
\end{align*}
\]

(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 \( \alpha_i \) to be \( L \) and \( \alpha_m \) to be \( H \), i.e.,
the former is \text{public} and the latter is \text{secret}.

Assuming that the programmer minimally labels the source program, only indicating those data that \text{must} be \text{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

\[
\begin{align*}
(\alpha_i \cup L) \cup L & \subseteq L & \text{line 5} \\
\alpha_i & \subseteq H & \text{line 6, for } h[i] \text{ in guard} \\
l_0 = \alpha_i \cup H & \cup \alpha_m \cup L & \text{PC label for checking if branch} \\
\alpha_i & \subseteq H & \text{line 6, for } h[i] \text{ in assignment} \\
l_0 \cup (H \cup \alpha_i) & \subseteq \alpha_m & \text{line 6, assignment} \\
L & \cup (\alpha_i \cup L) & \subseteq \alpha_i & \text{line 7}
\end{align*}
\]
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 \(T_i\) and \(T_f\) for the true and false branches of an if (following ORAM bank assignment), let \(T_{i,f}\) denote the 2-scs of \(T_i\) and \(T_f\). The differences between \(T_{i,f}\) 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.

When running 2-scs on traces, we view \(T_i\) 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, SAarr\((x, e)\). For example, suppose we have the program \(\text{skip} : o_x[y] := z\) 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 \(\text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch}\). For the purposes of running 2-scs, this trace consists of two characters: \((\text{Fetch} \circ)\), which corresponds to the statement \(\text{skip}\), and \((\text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch}\), 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 \(\text{if}(e, o \text{skip} : o_x[y] := z)\) and thus \(T_i = \text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch}\). Computing the 2-scs we find that \(T_i\) can be pre-padded with \(\text{Fetch} \circ \text{Fetch} \circ \text{Fetch} \circ \text{Fetch}\) while \(T_f\) can be post-padded with \(\text{Fetch}\). We can readily generate statements that correspond to both. For the second, we produce \(\text{skip}\). 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 SAarr\((x, e)\).

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

---

\[T_i := T_1 \downarrow T_3 \uparrow T_5 \quad | \quad T_f := T_2 \downarrow T_4 \uparrow T_6\]

Figure 8. Finding a short padding sequence using the greatest common subsequence algorithm. An example with two abstract traces \(T_i = [T_1; T_2; T_3; T_4; T_5]\) and \(T_f = [T_1; T_3; T_2; T_4]\). One greatest common subsequence as shown is \([T_1; T_2; T_4]\). A shortest common supersequence of the two traces is \(T_{i,f} = [T_1; T_3; T_2; T_3; T_4; T_5]\).

Because of the restrictions imposed by the type system, \(T_i\) and \(T_f\) will never contain loop patterns, (public) read-array or write patterns, or or-patterns.
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 (na"ïve \( O(n^3) \) 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 III. 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 necessary padding to ensure obliviousness.

A. Asymptotic Analysis

For the four programs we choose, Table II 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 \( \text{polylog } m \) 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.

### Table II

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dijkstra ((n = 100 \text{ nodes}))</td>
</tr>
<tr>
<td>2</td>
<td>K-means ((n = 100 \text{ data points}, k = 2, I = 1 \text{ iteration}))</td>
</tr>
<tr>
<td>3</td>
<td>Matrix multiplication ((n \times n \text{ matrix where } n = 40))</td>
</tr>
<tr>
<td>4</td>
<td>Matrix multiplication ((n \times n \text{ matrix where } n = 25))</td>
</tr>
<tr>
<td>5</td>
<td>Find max ((n = 100 \text{ elements in the array}))</td>
</tr>
<tr>
<td>6</td>
<td>Find max ((n = 10000 \text{ elements in the array}))</td>
</tr>
</tbody>
</table>

Memory accesses due to instruction fetches. Table II 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 our technical report [28]. 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 \( 1 \text{ to } 3 \text{ orders of magnitude} \) faster than the strawman for the chosen programs and parameters.

VI. RELATED WORK

We are the first to approach the problem of achieving privacy using ORAM from a programming languages theory.
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).

<table>
<thead>
<tr>
<th>Program</th>
<th>Memory accesses for data</th>
<th>Memory accesses for instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strawman</td>
<td>Opt 1</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$O(n^2 \log^2 n)$</td>
<td>$O(n^2 \log^2 n)$</td>
</tr>
<tr>
<td>K-means</td>
<td>$O(n \log^2 k)$</td>
<td>$O(n \log^2 k)$</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>$O(n^3 \log_2 n)$</td>
<td>$O(n^3 \log_2 n)$</td>
</tr>
<tr>
<td>Find max</td>
<td>$O(n \log^2 n)$</td>
<td>$O(n \log^2 n)$</td>
</tr>
</tbody>
</table>

perspective. Previously, the research community has largely considered generic oblivious program simulation (i.e., placing all variables in a single ORAM) \[12\] [\[17\]], 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 \[59\]. Several others have proposed custom data-oblivious algorithms \[11\] [\[20\]] 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 \[16\] [\[18\] [\[19\] [\[27\] [\[45\] [\[46\]]. 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 \[11\] [\[4\] [\[7\] [\[22\]] 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.

ACKNOWLEDGEMENTS

We thank Bobby Bhattacharjee and Mohit Tiwari for helpful technical discussions, and the anonymous reviewers for their feedback. This research was sponsored by NSF grants CNS-1111599 and CNS-0917098, and the US Army Research laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001.

REFERENCES

[22] Daniel Hedin and David Sands. Timing aware in-


[44] David Lie Chandramohan Thekkath, Mark Mitchell,


