GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation

Chang Liu,[†] Austin Harris,[‡] Martin Maas,[*] Michael Hicks,[†] Mohit Tiwari,[‡] and Elaine Shi[†]

[†] University of Maryland, College Park [‡] University of Texas at Austin [*] University of California, Berkeley

Abstract

This paper presents a new, co-designed compiler and architecture called GhostRider for supporting privacy preserving computation in the cloud. GhostRider ensures all programs satisfy a property called *memory-trace obliviousness* (MTO): Even an adversary that observes memory, bus traffic, and access times while the program executes can learn nothing about the program's sensitive inputs and outputs. One way to achieve MTO is to employ Oblivious RAM (ORAM), allocating all code and data in a single ORAM bank, and to also disable caches or fix the rate of memory traffic. This baseline approach can be inefficient, and so GhostRider's compiler uses a program analysis to do better, allocating data to non-oblivious, encrypted RAM (ERAM) and employing a scratchpad when doing so will not compromise MTO. The compiler can also allocate to multiple ORAM banks, which sometimes significantly reduces access times. We have formalized our approach and proved it enjoys MTO. Our FPGA-based hardware prototype and simulation results show that GhostRider significantly outperforms the baseline strategy.

1. Introduction

Cloud computing allows users to outsource both data and computation to third-party cloud providers, and promises numerous benefits such as economies of scale, easy maintenance, and ubiquitous availability. These benefits, however, come at the cost of giving up physical control of one's computing infrastructure and private data. Privacy concerns have held back government agencies and businesses alike from outsourcing their computing infrastructure to the public cloud [9, 52].

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694385

To protect the confidentiality of sensitive data in the cloud, thwarting software attacks alone is necessary but not sufficient. An attacker with physical access to the computing platform (e.g., an malicious insider or intruder) can launch various *physical attacks*, such as tapping memory buses, plugging in malicious peripherals, or using cold-(re)boots [26, 45]. Such physical attacks can uncover secrets even when the software stack is provably secure.

While memory encryption [19, 31, 47–49] can be used to hide the contents of memory from direct inspection, an adversary can still observe memory addresses transmitted over the memory bus. Perhaps surprisingly, the memory address trace is a side channel that can leak sensitive information, e.g., cryptographic keys [56]. To cryptographically obfuscate the memory access patterns, one can employ Oblivious RAM (ORAM) [20, 21], a cryptographic construction that makes memory address traces computationally indistinguishable from a random address trace. Encouragingly, recent theoretical breakthroughs [44, 46] have allowed ORAM memory controllers to be built [16, 35] – these turn DRAM into oblivious, block-addressable memory banks.

The simplest way to deploy ORAM is to implement a single, large ORAM bank that contains all the code and data (assuming that the client can use standard PKI to safely transmit the code and data to a remote secure processor). A major drawback of this baseline approach is efficiency: every single memory-block access incurs the ORAM penalty which is roughly (poly-)logarithmic in the size of the ORAM [21, 44]. In practice, this translates to almost $100\times$ additional bandwidth that, even with optimizations, incurs a $\sim 10\times$ latency cost per block [16, 35]. Another issue is that, absent any padding, the baseline approach reveals the total number of memory accesses made by a program, which can leak information about the secret inputs.

Memory Trace Oblivious (MTO) Program Execution Ultimately, the goal is to ensure that programs satisfy a property called *memory-trace oblivious* (MTO) execution, meaning that an adversary who observes memory contents, address traffic, and access times while the program executes can learn nothing about the program's sensitive inputs and outputs. While allocating code and data in ORAM is one approach toward achieving MTO,[1] our prior work [33] suggested that greater efficiency can be achieved by having the compiler take advantage of the fact that not all parts of a program's memory trace leak sensitive information. As an example, a program that sequentially scans through a sensitive array and computes the sum has a fixed, predictable memory access pattern that is independent of the sensitive array contents. In this case, it suffices to merely encrypt the array, instead of placing the array in ORAM. A further efficiency gain stems from placing data into different ORAM banks, which can now be smaller and in turn faster to access.

While our prior work lays out an initial vision of memory-trace oblivious program execution, this work is conceptual, rather than practical. In particular, we did not target a modern processor architecture, instead assuming unbounded resources and no caching.

1.1 Our Results and Contributions

In this paper, we make the first endeavor to bring the theory of MTO to practice. We design and build GhostRider, a hardware/software platform for provably secure, memory-trace oblivious program execution. Compiling to a realistic architecture while formally ensuring MTO raises interesting challenges in the compiler and type system design, and ultimately requires a co-operative re-design of the underlying processor architecture. Our contributions are:

New compiler and type system We build the first memory-trace oblivious compiler that emits target code for a realistic ORAM-capable processor architecture. The compiler must explicitly handle low-level resource allocation based on underlying hardware constraints, and while doing so is standard in non-oblivious compilers, achieving them while respecting the MTO property is non-trivial. Standard resource allocation mechanisms would fail to address the MTO property. For example, register allocation spills registers to the stack, thereby introducing memory events. Furthermore, caching serves memory requests from an on-chip cache, which suppresses memory events. If these actions are correlated with secret data, they can leak information. We introduce new techniques for resolving such challenges. In lieu of implicit caches we employ an explicit, on-chip *scratchpad*. Our compiler implements caching in software when its use does not compromise MTO.

To formally ensure the MTO property, we define a new type system for a RISC-style low-level assembly language. We show that any well-typed program in this assembly language will respect memory-trace obliviousness during execution. When starting from source programs that satisfy a standard information flow type system [13], our compiler generates type-correct, and therefore safe, target code. Specifically, we implement a type checker that can verify the type-correctness of the target code.

[1] In fact, ORAM allocation is not entirely sufficient: the *length* of an address trace can also reveal information, so additional steps are necessary.

```
void histogram(secret int a[],   // ERAM
               secret int c[]) { // ORAM (output)
  public int i;
  secret int t, v;
  for(i=0;i<100000;i++) // 100000 <= len(c)
    c[i]=0;
  i=0;
  for(i=0;i<100000;i++) { // 100000 <= len(a)
    v=a[i];
    if(v>0) t=v%1000;
       else t=(0-v)%1000;
    c[t]=c[t]+1; } }
```

Figure 1. Motivating source program.

Processor architecture for MTO program execution To enable an automated approach for efficient memory-trace oblivious program execution, we need new hardware features that are not readily available in existing ORAM-capable processor architectures [15, 17, 35]. GhostRider builds on the Phantom processor architecture [35] but exposes new features and knobs to the software. In addition to supporting a scratchpad, as mentioned above, the GhostRider architecture complements Phantom's ORAM support with *encrypted RAM* (ERAM), which is not oblivious and therefore more efficiently supports variables whose access patterns are not sensitive. Section 6 describes additional hardware-level contributions. We prototyped the GhostRider processor on a Convey HC2 platform [10] with programmable FPGA support. The GhostRider processor supports the RISC-V instruction set [51].

Implementation and Empirical Results Our empirical results are obtained through a combination of software emulation and experiments on an FPGA prototype. Our FPGA prototype supports one ERAM bank, one code ORAM bank, and one data ORAM bank. The real processor experiments demonstrate the feasibility of our architecture, while the software simulator allows us to test a range of configurations not limited by the constraints of the current hardware. In particular, the software simulator models multiple ORAM banks at a higher clock rate.

Our experimental results show that compared to the baseline approach of placing everything in a single ORAM bank, our compile-time static analysis achieves up to nearly an order-of-magnitude speedup for many common programs.

2. Architecture and Approach

This section motivates our approach and presents an overview of GhostRider's hardware/software co-design.

2.1 Motivating example

We wish to support a scenario in which a client asks an untrusted cloud provider to run a computation on the client's private data. For example, suppose the client wants the provider to run the program shown in Figure 1, which is

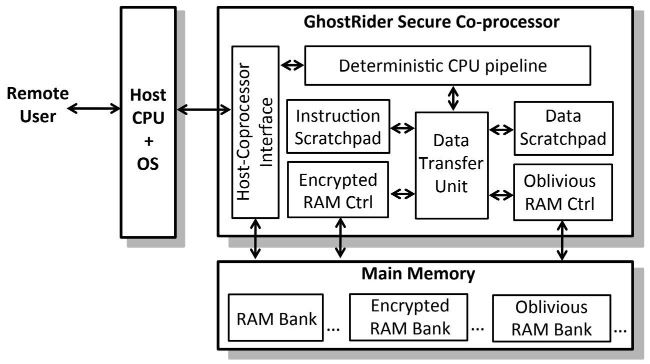


Figure 2. GhostRider architecture.

a simple histogram program written in a C-like source language. As input, the program takes an integer array a, and as output it modifies integer array parameter c. We assume both arrays have size 100,000. The function's code is straightforward, computing the histogram of the absolute values of integers modulo 1000 appearing in the input array. The client's security goal is *data confidentiality*: the cloud provider runs the program on input array a, producing output array c, but nevertheless learns nothing about the contents of either a or c. We express this goal by labeling both arrays with the qualifier `secret` (data labeled `public` is non-sensitive).

2.2 Threat model

The adversary has physical access to the machine(s) being used to run client computations. As in prior work that advocates the minimization of the hardware trusted computing base (TCB) [47–49], we assume that trust ends at the boundary of the secure processor. Off-chip components are considered *insecure*, including memory, system buses, and peripherals. For example, we assume the adversary can observe the contents of memory, and can observe communications on the bus between the processor and the memory. By contrast, we assume that on-chip components are secure. Specifically, the adversary cannot observe the contents of the cache, the register file, or any on-chip communications. Finally, we assume the adversary can make fine-grained timing measurements, and therefore can learn, for example, the gap between observed events. Analogous side channels such as power consumption are outside the scope of this paper, and have received orthogonal treatment in related work [29].

2.3 Architectural Overview

As mentioned in the introduction, one way to defend against such an adversary is to place all data in a single (large) ORAM; e.g., for the program in Figure 1 we place the arrays a and c in ORAM. Unfortunately this *baseline* approach is not only expensive, but also leaks information through the total number of ORAM accesses (if the access trace is not padded to a value that is independent of secret data). We now provide an architectural overview of GhostRider (Figure 2) and contrast it with this baseline.

Joint ORAM-ERAM memory system In the GhostRider architecture, main memory is split into three types—normal (unencrypted) memory (RAM), encrypted memory (ERAM), and oblivious RAM (ORAM)—with one or more (logical) banks of each type comprising the system's physical memory. The differentiation of memory into banks allows a compiler to place only arrays with sensitive access patterns inside the more expensive ORAM banks, while keeping the remaining data in the significantly faster RAM or ERAM banks. For example, notice that in the program in Figure 1 the array a is always accessed sequentially while access patterns to the array c can depend on secret array contents. Therefore, our GhostRider compiler can place the array a inside an ERAM bank, and place the array c inside an ORAM bank. The program accesses different memory banks at the level of blocks using instructions that specify the bank and a block-offset within the bank (after moving data to on-chip memory as described below). Our hardware prototype fixes block sizes to be 4KB for both ERAM and ORAM banks (which is not an inherent limitation of the hardware design).

Software-directed scratchpad As mentioned earlier, cache hit and miss behavior can lead to differences in the observable memory traces. To prevent such cache-channel information leakage, the GhostRider architecture turns off implicit caching, and instead offers software-directed scratchpads for both instructions and data. These scratchpads are mapped into the program's address space so that the compiler can generate code to access them explicitly, and thereby avoid information leaks. For example, the indices of array a in Figure 1 are deterministic; they do not depend on any secret input. As such, it is safe to use the scratchpad to cache array a's accesses. The compiler generates code to check whether the relevant block is in the scratchpad, and if not loads the block from memory. On the other hand, all accesses to array c depend on the secret input a, so a memory request will always be issued independent of whether the requested block is in the scratchpad or not.

Deterministic Processor Pipeline To avoid timing-channel leakage, our pipelined processor ensures that instruction timings are deterministic. We do not use dynamic branch prediction and fix variable-duration instructions, such as division, to take the worst-case execution time, and disable concurrent execution of other instructions.

Initialization We design the oblivious processor and memory banks as a *co-processor* that runs the application natively (i.e., without an OS) and is connected to a networked host computer that can be accessed remotely by a user. We assume that the secure co-processor has non-volatile memory for storing a long-term public key (certified using PKI), such that the client can securely ship its encrypted code and data to the remote host, and initialize execution on the secure co-processor. Implementing the secure attestation is standard [25], and we leave it to future work.

m, n	$\in$ $\mathbb{Z}$		$o_1, ..., o_n \in$ **ORAMbanks**
k	$\in$ **Block IDs**		$r \in$ **Registers**
l	$\in$ **Labels** $= \{D, E\} \cup$ **ORAMbanks**		
ι	$::=$ **ldb** $k \leftarrow l[r]$		load block to scratchpad
	$\mid$ **stb** k		store block to memory
	$\mid$ $r \leftarrow$ **idb** k		retrieve the block ID
	$\mid$ **ldw** $r_1 \leftarrow k[r_2]$		load a scratchpad val. to reg.
	$\mid$ **stw** $r_1 \rightarrow k[r_2]$		store a reg. val. to scratchpad
	$\mid$ $r_1 \leftarrow r_2\ aop\ r_3$		compute an operation
	$\mid$ $r_1 \leftarrow n$		assign a constant to a register
	$\mid$ **jmp** n		(relative) jump
	$\mid$ **br** $r_1\ rop\ r_2 \hookrightarrow n$		compare and branch
	$\mid$ **nop**		empty operation
I	$::= \iota \mid I; \iota$		instruction sequence

Figure 3. Syntax for $\mathcal{L}_T$ language, comprising (1) **ldb** and **stb** *i*nstructions that move data blocks between scratchpad and a specific ERAM or ORAM bank, and (2) scratchpad-to-register moves and standard RISC instructions.

```
; v=a[i]
1   $t_1 \leftarrow r_i$ div $size_{blk}$
2   $t_2 \leftarrow r_i$ mod $size_{blk}$
3   ldb $k_1 \leftarrow E[t_1]$
4   ldw $r_v \leftarrow k_1[t_2]$
; if(v>0) t= ...
5   br $r_v \leq 0 \hookrightarrow 3$
6   $r_t \leftarrow r_v$ % 1000
7   jmp 3
; else t= ...
8   $t_1 \leftarrow 0 - r_v$
9   $r_t \leftarrow t_1$ % 1000
; c[t]=c[t]+1
10  $t_1 \leftarrow r_t \gg 9$
11  $t_2 \leftarrow r_t$ & 511
12  ldb $k_2 \leftarrow O[t_1]$
13  ldw $t_3 \leftarrow k_2[t_2]$
14  $t_4 \leftarrow t_3 + 1$
15  stw $t_4 \rightarrow k_2[t_2]$
16  stb $k_2$
```

Figure 4. $\mathcal{L}_T$ code implementing (part of) Figure 1

3. Target language

This section presents a small formalization of GhostRider's instruction set, which we call $\mathcal{L}_T$. The next section presents a type system for this language that guarantees security, and the following section describes our compiler from a C-like source language to well-typed $\mathcal{L}_T$ programs.

3.1 Instruction set

The core instructions of $\mathcal{L}_T$ are in the style of RISC-V [51], our prototype's instruction set, and are formalized in Figure 3. We define *labels* l that distinguish the three kinds of main memory: D for normal (D)RAM, E for ERAM, and o_i for ORAM. For the last, the i identifies a particular ORAM bank. We can view each label as defining a distinct address space.

The instruction **ldb** $k \leftarrow l[r]$ loads a block from memory into the scratchpad.[2] Here, l is the address space, r is a register containing the address of the block to load from within that address space, and k is the scratchpad block identifier. Our formalism refers to scratchpad blocks by their identifier, treating them similarly to registers. Our architecture remembers the address space and block address within that address space that the scratchpad block was loaded from so that writebacks, via the **stb** k instruction, will go to the original location. We enforce this one-to-one mapping to avoid information leaks via write-back from the scratchpad (e.g., that where a scratchpad block is written to, in memory, could reveal information about a secret, or that the effect of a write could do so, if blocks are aliased).

[2] In our hardware prototype the scratchpad is mapped into addressable memory, so this instruction and its counterpart, **stb**, are implemented as data transfers. In addition, the compiler implements **idb**. We model them in $\mathcal{L}_T$ explicitly for simplicity; see Section 6 for implementation details.

To access values from the scratchpad, we have scratchpad-load and scratchpad-store instructions. The scratchpad-load instruction loads a word from a scratchpad block, having the form **ldw** $r_1 \leftarrow k[r_2]$. Assuming register r_2 contains n, this instruction loads the n-th word in block k into register r_1 (notice that we use word-oriented addressing, not byte-oriented). The scratchpad-store instruction is similar, but goes in the reverse direction. The instruction $r \leftarrow$ **idb** k retrieves the block offset of a scratchpad block k.

We have two kinds of assignment instructions, one in the form of $r_1 \leftarrow r_2\ aop\ r_3$, and the other in the form of $r \leftarrow n$. In $\mathcal{L}_T$ we only model integer arithmetic operations, such as addition, subtraction, multiplication, division, and modulus.

Jumps and branches use relative addressing. The jump instruction **jmp** n bumps the program counter by n instructions (where n can be negative). Branches, having the form **br** $r_1\ rop\ r_2 \hookrightarrow n$, will compare the contents of r_1 and r_2 using rop, and will bump the pc by n if the comparison result is true. An instruction sequence I is defined to be a sequence of instructions concatenated using a logical operation ;. We overload ; to operate over two instruction sequences such that $I; (I'; \iota) \triangleq (I; I'); \iota$ and $I_1; I_2; I_3 \triangleq (I_1; I_2); I_3$.

Note that our formalism does not model the instruction scratchpad; essentially it assumes that all code is loaded on-chip prior to the start of its execution. Section 5 discusses how the instruction scratchpad is used in practice.

3.2 Example

Figure 4 shows $\mathcal{L}_T$ code that corresponds to the body of the second `for` loop in the source program from Figure 1. We write r_X for a register corresponding to variable X in the source program (for simplicity) and write t_i for $i \in \{1, 2, ...\}$ for temporary registers. In the explanation we refer to the names of variables in the source program when describing what the target program is computing.

The first four lines load the `i`th element of array `a` into `v`. Line 1 computes the address of the block in memory that contains the `i`th element of array `a` and line 2 computes the offset of the element within that block. Here $size_{blk}$ is the size of each block, which is an architecture constant.

Line 3 then loads the block from ERAM, and line 4 loads the appropriate value from the loaded block into `v`.

The next five lines implement the conditional. Line 5 jumps three instructions forward if `v` is *not* greater than 0, else it falls through to line 6, which computes `t`. Line 7 then jumps past the else branch, which begins on line 8, which negates `v` to make it positive before computing `t`.

The final seven lines increment `c[t]`. Lines 10–13 are analogous to lines 1–4; they compute the address of `t`th element of array `c` and load it into temporary t_3. Notice that this time the block is loaded from ORAM, not ERAM. Line 14 increments the temporary; line 15 stores it back to the block in the scratchpad; and line 16 stores the entire block back to ORAM.

4. Security by typing

This section presents a type system[3] for $\mathcal{L}_T$ that guarantees programs obey the strong *memory trace obliviousness* (MTO) security property.

4.1 Memory Trace Obliviousness

Memory trace obliviousness is a noninterference property that also considers the address trace, rather than just the initial and final contents of memory [43]. MTO's definition relies on the notion of *low equivalence* which relates memories whose RAM contents are identical. We formally define this notion below, using the following formal notation:

$$\begin{array}{rcl} M & \in & \textbf{Addresses} \to \textbf{Blocks} \\ a & \in & \textbf{Addresses} = \textbf{Labels} \times \textbf{Nat} \\ b & \in & \textbf{Blocks} = \textbf{Nat} \to \mathbb{Z} \end{array}$$

We model a *memory* M as a map from addresses to blocks, where an address is a pair consisting of a label l (corresponding to an ORAM, ERAM, or RAM bank, as per Figure 3) and an address n in that bank. A block is modeled as a map from an address n to a (integer) value. Here is the definition of memory low-equivalence:

DEFINITION 1 (Memory low equivalence). *Two memories* M_1, M_2 *are* low equivalent, *written* $M_1 \sim_L M_2$, *if and only if for all* n *such that* $0 \leq n < size(D)$ *we have* $M_1(D,n) = M_2(D,n)$.

The definition states that memories M_1 and M_2 are low equivalent when only the RAM bank's values of the memories are the same, but all of the other values could differ.

Intuitively, memory trace obliviousness says two things given two low-equivalent memories. First, if the program will terminate under one memory, then it will terminate under the other. Second, if the program will terminate and lead to a trace t under one memory, then it will do so under the other memory as well while also finishing with low-equivalent memories.

To state this intuition precisely, we need a formal definition of a $\mathcal{L}_T$ execution, which we give as an operational semantics. The semantics is largely standard, and can be found in the technical report [32]. The key judgment has the form $I \vdash (R, S, M, pc) \longrightarrow_t (R', S', M', pc')$, which states that program I, with a register file R, a (data) scratchpad S, a memory M, and a program counter pc, executes some number of steps, producing memory trace t and resulting in a possibly modified register file R', scratchpad S', memory M', and program counter pc'.

DEFINITION 2 (Memory trace obliviousness). *A program* I *is* memory trace oblivious *if and only if for all memories* $M_1 \sim_L M_2$ *we have* $I \vdash (R_0, S_0, M_1, 0) \longrightarrow_{t_1} (R'_1, S'_1, M'_1, pc_1)$*, and* $I \vdash (R_0, S_0, M_2, 0) \longrightarrow_{t_2} (R'_2, S'_2, M'_2, pc_2)$*, and* $|t_1| = |t_2|$ *implies* $t_1 \equiv t_2$ *and* $M'_1 \sim_L M'_2$.

Here R_0 is a mapping that maps every register to 0, and S_0 maps every address to a all-0 block. Traces t consist of reads/writes to RAM (both address and value) and ERAM (just the address), accesses to ORAM (just the bank), and instruction fetches. For the last we only model that a fetch happened, not what instruction it is, as we assume code will be stored in a scratchpad on chip. We write $t_1 \equiv t_2$ to say that traces t_1 and t_2 are *indistinguishable* to the attacker; i.e., they consist of the same events in the same order. Our formalism models every instruction as taking *unit* time to execute – thus the trace event also models the time taken to execute the instruction. On the real GhostRider architecture, each instruction takes *deterministic but non-uniform* time; as this difference is conceptually easy to handle (by accounting for instruction execution times in the compiler), we do not model it formally, for simplicity (see Section 5).

4.2 Typing: Preliminaries

Now we give a type system for $\mathcal{L}_T$ programs and prove that type correct programs are MTO.

Symbolic values To ensure that the execution of a program cannot leak information via its address trace, we statically approximate what events a program can produce. An important determination made by the type system is when a secret variable can be stored in ERAM—because the address trace will leak no information about it—and when it must be accessed in ORAM. As an example, suppose we had the source program

```
if(s) then x[i]=1 else x[i]=2;
```

If `x` is secret but stored in RAM, then the value in `x[i]` after running this program will leak the contents of secret variable `s`. We could store `x` in ORAM to avoid this problem, but this is unnecessary: both branches will modify the same element of `x`, so encrypting the content of `x` is enough to prevent the address trace from leaking information about `s`. The type

[3] For space reasons, the type system is simplified from the full version, given in the extended technical report [32]. The full system models a stack to support function calls, which we discuss briefly at the end of Section 5.

$$
\begin{array}{llll}
\text{Sym. vals.} & sv & \in \textbf{SymVals} = n \mid ? \mid sv_1\ aop\ sv_2 \mid \mathtt{M}_l[k, sv] \\
\text{Sym. Store} & Sym & \in \textbf{Registers} \cup \textbf{Block IDs} \to \textbf{SymVals} \\
\text{Sec. Labels} & \ell & \in \textbf{SecLabels} = \mathtt{L} \mid \mathtt{H} \\
\text{Label Map} & \Upsilon & \in (\textbf{Registers} \to \textbf{SecLabels}) \\
 & & \cup\ (\textbf{Block IDs} \to \textbf{Labels})
\end{array}
$$

$\boxed{sv_1 \equiv sv_2}$

$$\frac{\vdash_{safe} sv_1 \quad \vdash_{safe} sv_2 \quad sv_1 = sv_2}{sv_1 \equiv sv_2}$$

$\boxed{\vdash_{safe} sv}$

$$\frac{l = D \quad \vdash_{safe} sv}{\vdash_{safe} \mathtt{M}_l[k, sv]} \qquad \vdash_{safe} n$$

$$\frac{\vdash_{safe} sv_1 \quad \vdash_{safe} sv_2}{\vdash_{safe} sv_1\ aop\ sv_2}$$

$\boxed{\text{Auxiliary Functions}}$

$$select(l, a, b, c) = \begin{cases} a & \text{if } l = D \\ b & \text{if } l = E \\ c & \text{if otherwise.} \end{cases}$$

$$slab(l) = select(l, \mathtt{L}, \mathtt{H}, \mathtt{H})$$

$$ite(x, a, b) = \begin{cases} a & \text{if } x \text{ is true.} \\ b & \text{otherwise.} \end{cases}$$

$\boxed{\vdash_{const} sv}$

$$\vdash_{const} n \qquad \vdash_{const} ?$$

$$\frac{\vdash_{const} sv_1 \quad \vdash_{const} sv_2}{\vdash_{const} sv_1\ aop\ sv_2}$$

$\boxed{\vdash_{const} Sym}$

$$\frac{\forall r.\ \vdash_{const} Sym(r) \quad \forall k.\ \vdash_{const} Sym(k)}{\vdash_{const} Sym}$$

Figure 5. Symbolic values, labels, auxiliary judgments and functions

system can identify this situation by *symbolically* tracking the contents of the registers, blocks, etc.

To do this, the type rules maintain a *symbolic store* Sym, which is a map from register and block IDs to symbolic values. Figure 5 defines symbolic values sv, which consist of constants n, (symbolic) arithmetic expressions, values loaded from memory $\mathtt{M}_l[k, sv]$, and unknowns ?. Most interesting is memory values, which represent the address of a loaded value: l indicates the memory bank it was loaded from, sv corresponds to the offset (i.e., the block number) within that bank, and k is the scratchpad block into which the memory block is loaded.[4]

The type rules also make use of a *label map* Υ mapping registers to security labels and block IDs to (memory) labels; the latter tracks the memory bank from which a scratchpad block was loaded.

The figure defines several judgments; the form of each judgment is boxed. The first defines when two symbolic values can be deemed equivalent, written $sv_1 \equiv sv_2$: they must be syntactically identical and safe static approximations. The latter is defined by the judgment $\vdash_{safe} sv$, which accepts constants, memory accesses to RAM involving safe indexes, and arithmetic expressions involving safe values. Judgement $\vdash_{const} sv$ says that symbolic value sv is not a memory value. That is, sv is either a constant, a ?, or a binary expression not involving memory values. Further, for a symbolic store Sym, if all the symbolic values that it maps to can be accepted by $\vdash_{const} sv$, then we have $\vdash_{const} Sym$. The latter judgment is needed when checking conditionals.

[4] In actual traces t, the block number k is not visible; we track it symbolically to model the scratchpad's contents, in particular to ensure that the same memory block is not loaded into two different scratchpad blocks.

$$
\begin{array}{lll}
\text{Trace Pats.} & T ::= & \textbf{read}(l, k, sv) \mid \textbf{write}(l, k, sv) \mid \textbf{F} \mid o \\
 & \mid & T_1 @ T_2 \mid T_1 + T_2 \mid \textbf{loop}(T_1, T_2)
\end{array}
$$

$$\frac{sv_1 \equiv sv_2}{\textbf{read}(l, k, sv_1) \equiv \textbf{read}(l, k, sv_2)} \qquad o \equiv o \qquad \frac{T_1 \equiv T_2}{T_2 \equiv T_1}$$

$$\frac{sv_1 \equiv sv_2}{\textbf{write}(l, k, sv_1) \equiv \textbf{write}(l, k, sv_2)} \qquad \textbf{F} \equiv \textbf{F}$$

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

Figure 6. Trace patterns and their equivalence

Finally, we give three auxiliary functions used in the type system. Based on whether l is D, E, or an ORAM bank, function $select(l, a, b, c)$ returns a, b, or c respectively. Function $slab(\cdot)$ maps a normal label l to a security label ℓ, which is either `L` or `H`. The label `H` classifies encrypted memory—any ORAM bank and ERAM—while label `L` classifies RAM. These two labels form the two-point lattice with $\mathtt{L} \sqsubset \mathtt{H}$. Note that `L` is equivalent to the `public` label used in Figure 1, and `H` is equivalent to `secret`. Finally, function $ite(x, a, b)$ returns a if x is true, and returns b if x is false.

Trace patterns Figure 6 defines *trace patterns* T, which statically approximate traces t. The first line in the definition of T defines single events. The first two indicate reads and writes to RAM or ERAM; they reference the memory bank, block identifier in the scratchpad, and a symbolic value corresponding to the block *address* (not the actual value) read or written. Pattern **F** corresponds to a non memory-accessing instruction. The next pattern indicates a read or write from ORAM bank o: this bank is the trace event itself because the adversary cannot determine whether an access is a read or a write, or which block within the ORAM is accessed. Trace pattern $T_1 @ T_2$ is the pattern resulting from the concatenation of patterns T_1 and T_2. Pattern $T_1 + T_2$ represents *either* T_1 or T_2, and is used to type conditionals. Finally, pattern $\textbf{loop}(T_1, T_2)$ represents zero or more loop iterations where the guard's trace is T_1 and the body's trace is T_2.

Trace pattern equivalence $T_1 \equiv T_2$ is defined in Figure 6. In this definition, reads are equivalent to other reads accessing exactly the same location; the same goes for writes. Two ORAM accesses to the same ORAM bank are obviously treated as equivalent. Sum patterns specify possibly different trace patterns, and loop patterns do not specify the number of iterations; as such we cannot determine their equivalence statically. The concatenation operator @ is associative with respect to equivalence.

4.3 Type rules

Figure 7 defines the security type system for $\mathcal{L}_T$. The figure is divided into three parts.

Instructions Judgment $\ell \vdash \iota : \langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle; T$ is used to type instructions ι. Here, ℓ is the *security context*, used in the standard way to prevent implicit flows. The rules are *flow sensitive*: The judgement says that instruction ι has a type $\langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle$, and generates trace pattern T. Informally, we can say by executing ι, a state corresponding to security type $\langle \Upsilon, Sym \rangle$ will be changed to have type $\langle \Upsilon', Sym' \rangle$.

Rule T-LOAD types load instructions. The first premise ensures that the contents of register r, the indexing register, are not leaked by the operation. In particular, the loaded memory bank l must either be ORAM, or else the register r may only contain public data (from RAM). In the latter case, there is no issue with leaking r, and in the former case r will not be leaked indirectly by the address of the loaded memory since it is stored in ORAM. The final two premises determine the final trace pattern. When the memory bank l is D or E, then the trace pattern indicates a read event from the appropriate block and address. When reading from an ORAM bank the event is just that bank itself. The other premises in the rule update Υ to map the loaded block k to the label of the memory bank, and update Sym to track the address of the block.

We defer discussion of rule T-STORE for the moment, and look at the next three rules, T-LOADW, T-STOREW, T-IDB, which are used to load and store values related blocks in the scratchpad. The first two rules resemble standard information flow rules. The second premise of T-LOADW is similar to the first premise of T-LOAD in preventing an indirect leak of index register r_2, which would occur if the label of r_2 was $\mathtt{H}$ but the label of k was $\mathtt{L}$. Likewise, the premise of T-STOREW prevents leaking the contents of r_1 and r_2 into the stored block, and also prevents an implicit flow from ℓ (the security context). As such, these two rules ensure that a block k with label ℓ never contains information from memory labeled ℓ' such that $\ell' \sqsupset \ell$. The remaining premises of Rule T-LOADW flow-sensitively track the label and symbolic value of the loaded register. In particular, they set the label of r_1 to be that of the block loaded, and the symbolic value of r_1 to be the address of the loaded value in memory. T-STOREW changes neither Υ nor Sym: even though the content of the scratchpad has changed, its memory label and its address in memory has not. Both rules emit trace pattern $\mathbf{F}$ as the operations are purely on-chip. We emit this event to account for the time taken to execute an instruction; assuming uniform times for instructions and memory accesses, MTO executions will also be free of timing channels.

Returning to rule T-STORE, we can see that the store takes place unconditionally—no constraints on the labels of the memory or block must be satisfied. This is because the other type rules ensure that all blocks k never contain information higher than their security label ℓ, and thus the block can be written straight to memory having the same security label. That said, information could be leaked through the memory trace, so the emitted trace pattern will differ depending on the label of the block: If the label is D or E then the trace pattern will be a write event, and otherwise it will be the appropriate ORAM event. Leaks via the memory trace are then prevented by T-IF and T-LOOP, discussed shortly.

Rule T-IDB is similar to rule T-LOADW. For the third premise, if l is either D or E, the block k has a public address, and thus the value assigned to register r is public; otherwise, when l is an ORAM bank, the register r is secret.

Rule T-BOP types binary operations, updating the security label of the target register to be the join of labels of the source registers. Rule T-ASSIGN gives the target register label $\mathtt{L}$ as constants are not secret. Rules T-NOP is always safe and has no effect on the symbolic store or label environment. All of these operations occur on-chip, and so have pattern $\mathbf{F}$. Finally, rule T-SEQ types instruction sequences by composing the symbolic maps, label environments, and traces in the obvious way.

Branching Rules T-IF and T-LOOP consider structured control flow. Rule T-IF deals with instruction sequences of the form of $I = \iota_1; I_t; \iota_2; I_f$, where ι_1 is a branching instruction deciding, ι_2 is a jump instruction jumping over the false branch, and I_t and I_f are the true and false branches respectively; the relative offsets n_1 and n_2 are based on the length of these code sequences. We require both branches to have the same type, i.e. $\langle \Upsilon, Sym \rangle \rightarrow \langle \Upsilon', Sym' \rangle$, as for the sequence I itself.

When the security context is high, i.e. $\ell = \mathtt{H}$, or when the if-condition is private, i.e. $\Upsilon(r_1) \sqcup \Upsilon(r_2) = \mathtt{H}$, then ℓ' will be $\mathtt{H}$ and we impose three restrictions. First, both of the blocks I_t and I_f must have equivalent trace patterns. (The trace of the true branch is $T_1 @ \mathbf{F}$ where T_1 covers I_t and $\mathbf{F}$ covers the jump instruction ι_2.) Second, if the security context is public, i.e. $\ell = \mathtt{L}$, then we restrict $\vdash_{const} Sym$ to enforce $Sym(r)$ does not map to memory values. The reason is that in a public context, two equivalent symbolic memory values may refer to two different concrete values, since the memory region D can be modified. Third, for any register r, its value after taking either branch must be the same, or the register r must have a high security label (i.e. $\Upsilon'(r) = \mathtt{H}$). So if $\Upsilon'(r) = \mathtt{L}$, the type system enforces that its symbolic values on the two paths are equivalent, i.e. $Sym'(r) \equiv Sym'(r)$, which only requires $\vdash_{safe} Sym'(r)$.

The final premise for rule T-IF states that the sequence's trace pattern T is either $\mathbf{F} @ T_1 @ \mathbf{F}$ when both branches' patterns must be equal, or else is an or-pattern involving the trace T_2 from the else branch.

$$\boxed{\text{Instructions}}$$

$$\text{T-LOAD}\ \frac{\begin{array}{c} l \notin \textbf{ORAMbanks} \Rightarrow \Upsilon(r) = \mathtt{L} \\ \Upsilon' = \Upsilon[k \mapsto l] \qquad \mathit{Sym}' = \mathit{Sym}[k \mapsto \mathit{Sym}(r)] \\ T_0 = \textbf{read}(l, k, \mathit{Sym}(r)) \qquad T = \mathit{select}(l, T_0, T_0, l) \end{array}}{\ell \vdash \textbf{ldb}\ k \leftarrow l[r] : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T}$$

$$\text{T-STORE}\ \frac{\begin{array}{c} \mathit{Sym}(k) = sv \qquad \Upsilon(k) = l \\ T_0 = \textbf{write}(l, k, sv) \qquad T = \mathit{select}(l, T_0, T_0, l) \end{array}}{\ell \vdash \textbf{stb}\ k : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon, \mathit{Sym} \rangle; T}$$

$$\text{T-LOADW}\ \frac{\begin{array}{c} l = \Upsilon(k) \qquad \Upsilon(r_2) \sqsubseteq \mathit{slab}(l) \\ \Upsilon' = \Upsilon[r_1 \mapsto \mathit{slab}(l)] \qquad sv = \mathtt{M}_l[k, \mathit{Sym}(r_2)] \\ \mathit{Sym}' = \mathit{Sym}[r_1 \mapsto sv] \end{array}}{\ell \vdash \textbf{ldw}\ r_1 \leftarrow k[r_2] : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; \mathbf{F}}$$

$$\text{T-STOREW}\ \frac{\ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathit{slab}(\Upsilon(k))}{\ell \vdash \textbf{stw}\ r_1 \to k[r_2] : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon, \mathit{Sym} \rangle; \mathbf{F}}$$

$$\text{T-IDB}\ \frac{\begin{array}{c} \mathit{Sym}(k) = sv \qquad \Upsilon(k) = l \\ \Upsilon' = \Upsilon[r \mapsto \mathit{select}(l, \mathtt{L}, \mathtt{L}, \mathtt{H})] \qquad \mathit{Sym}' = \mathit{Sym}[r \mapsto sv] \end{array}}{\ell \vdash r \leftarrow \textbf{idb}\ k : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; \mathbf{F}}$$

$$\text{T-BOP}\ \frac{\begin{array}{c} \ell' = \Upsilon(r_2) \sqcup \Upsilon(r_3) \qquad \Upsilon' = \Upsilon[r_1 \mapsto \ell'] \\ sv = \mathit{Sym}(r_2)\ \mathit{aop}\ \mathit{Sym}(r_3) \qquad \mathit{Sym}' = \mathit{Sym}[r_1 \mapsto sv] \end{array}}{\ell \vdash r_1 \leftarrow r_2\ \mathit{aop}\ r_3 : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; \mathbf{F}}$$

$$\text{T-ASSIGN}\ \frac{\Upsilon' = \Upsilon[r \mapsto \mathtt{L}] \qquad \mathit{Sym}' = \mathit{Sym}[r \mapsto n]}{\ell \vdash r \leftarrow n : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; \mathbf{F}}$$

$$\text{T-NOP}\ \ \ell \vdash \textbf{nop} : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon, \mathit{Sym} \rangle; \mathbf{F}$$

$$\text{T-SEQ}\ \frac{\begin{array}{c} \ell \vdash I : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T_1 \\ \ell \vdash \iota : \langle \Upsilon', \mathit{Sym}' \rangle \to \langle \Upsilon'', \mathit{Sym}'' \rangle; T_2 \end{array}}{\ell \vdash I; \iota : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon'', \mathit{Sym}'' \rangle; T_1 @ T_2}$$

$$\boxed{\text{Branching}}$$

$$\text{T-IF}\ \frac{\begin{array}{c} I = \iota_1; I_t; \iota_2; I_f \qquad |I_t| = n_1 - 2 \qquad |I_f| + 1 = n_2 \\ \iota_1 = \textbf{br}\ r_1\ \mathit{rop}\ r_2 \hookrightarrow n_1 \qquad \iota_2 = \textbf{jmp}\ n_2 \\ \ell' = \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \\ \ell' \vdash I_t : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T_1 \\ \ell' \vdash I_f : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T_2 \\ \ell' = \mathtt{H} \Rightarrow \left\{ \begin{array}{l} T_1 @ \mathbf{F} \equiv T_2 \ \wedge \\ \ell = \mathtt{L} \Rightarrow \vdash_{const} \mathit{Sym} \ \wedge \\ \forall r. \Upsilon'(r) = \mathtt{L} \Rightarrow \vdash_{safe} \mathit{Sym}'(r) \end{array} \right\} \\ T = \mathit{ite}(\ell' = \mathtt{H},\ \mathbf{F} @ T_1 @ \mathbf{F},\ \mathbf{F} @ ((T_1 @ \mathbf{F}) + T_2)) \end{array}}{\ell \vdash I : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T}$$

$$\text{T-LOOP}\ \frac{\begin{array}{c} I = I_c; \iota_1; I_b; \iota_2 \\ |I_b| = n_1 - 2 \qquad |I_c| + n_1 = 1 - n_2 \\ \iota_1 = \textbf{br}\ r_1\ \mathit{rop}\ r_2 \hookrightarrow n_1 \qquad \iota_2 = \textbf{jmp}\ n_2 \\ \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L} \\ \ell \vdash I_c : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T_1 \\ \ell \vdash I_b : \langle \Upsilon', \mathit{Sym}' \rangle \to \langle \Upsilon, \mathit{Sym} \rangle; T_2 \end{array}}{\ell \vdash I : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; \textbf{loop}(T_1, T_2)}$$

$$\boxed{\text{Subtyping}}$$

$$\text{T-SUB}\ \frac{\begin{array}{c} \ell \vdash \iota : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T \\ \Upsilon' \preceq \Upsilon'' \qquad \mathit{Sym}' \preceq \mathit{Sym}'' \end{array}}{\ell \vdash \iota : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon'', \mathit{Sym}'' \rangle; T}$$

$$\text{S-LABEL}\ \frac{\begin{array}{c} \forall r. \Upsilon(r) \sqsubseteq \Upsilon'(r) \\ \forall k. \Upsilon(k) = \Upsilon'(k) \end{array}}{\Upsilon \preceq \Upsilon'}$$

$$\text{S-SYM}\ \frac{\begin{array}{c} \forall r. \mathit{Sym}'(r) = ? \vee \mathit{Sym}(r) = \mathit{Sym}'(r) \\ \forall k. \mathit{Sym}'(k) = ? \vee \mathit{Sym}(k) = \mathit{Sym}'(k) \end{array}}{\mathit{Sym} \preceq \mathit{Sym}'}$$

Figure 7. Security Type System for $\mathcal{L}_T$

Rule T-LOOP imposes structural requirements on I similar to T-IF. The premise $\ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$ implies two restrictions. On the one hand, $\ell \sqsubseteq \mathtt{L}$, prevents any loop from appearing in a secret if-statement, because otherwise the number of loop iterations may leak information about which branch is taken. On the other hand, $\Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L}$ implies that the loop condition must be public, or otherwise, similarly, the number of iterations would leak secret information about r_1 and/or r_2.

Subtyping Finally, rule T-SUB supports subtyping on the symbolic store and the label map. For the first, a symbolic store Sym can approximated by a store Sym' that either agrees on the symbolic values mapped to be Sym or maps them to ?. For the second, a register's security label can be approximated by one higher in the lattice; block labels may not change. Subtyping is important for typing join points after branches or loops. For example, if a conditional assigned a register r the value 1 in the true branch but assigned r to 2 in the false branch, we would use subtyping to map r to ? to ensure that the symbolic store at the end of both branches agrees, as required by T-IF.

4.4 Security theorem

All well-typed programs are memory-trace oblivious:

THEOREM 1. *Given I, Υ, and Sym, if there exists some Υ', Sym' and T such that $\mathtt{L} \vdash I : \langle \Upsilon, \mathit{Sym} \rangle \to \langle \Upsilon', \mathit{Sym}' \rangle; T$, where $\forall r. \mathit{Sym}(r) = ?$ and $\Upsilon(r) = L$ and $\forall k. \mathit{Sym}(k) = ?$ and $\Upsilon(k) = D$ then program I is memory-trace oblivious.*

The proof can be found in our technical report [32].

5. Compilation

We have developed a compiler from an imperative, C-like source language, which we call $\mathcal{L}_S$, to $\mathcal{L}_T$. Our compiler is implemented in about 7600 lines of Java, with roughly 400 LoC dedicated to the parser, 700 LoC to the type checker, 3500 LoC to the compiler/optimizer, 950 LoC to the code generator, and the remainder to utility functions. This section informally describes our compilation approach.

5.1 Source Language

Syntax An $\mathcal{L}_S$ program is a collection of (possibly mutually recursive) functions and a collection of (possibly mutually recursive) type definitions. A type definition is simply a mapping of a type name to a type where types are either natural numbers, arrays, or pointers to records (i.e., C-style `structs`). Each type is annotated with a security label which is either `secret` or `public` indicating whether the data should be visible/inferrable by the adversary or not.

A function consists of a sequence of statements s which are either no-ops, variable assignments, array assignments, conditionals, while loops, or returns. As usual, conditional branches and loop bodies may consist of one or more statements. Expressions e appearing in statements (e.g., in assignments) consist of variables x, arithmetic ops $e_1\ aop\ e_2$, array reads $e[e]$, and numeric constants n. Variables may hold any data other than functions (i.e., there are no function pointers). Guards in conditionals and while loops consist of predicates involving relational operators.

Typing $\mathcal{L}_S$ programs are type checked before they are compiled. We do this using an information flow-style type system (cf. the survey of Sabelfeld and Myers [43]). As is standard, the type system prevents explicit flows and implicit flows. In particular, it disallows assignments like `p = s` where `p` is a public variable and `s` is a secret variable, and disallows conditionals like `if (s == 0) then p = 0 else p = 1`, which leaks information about `s` since after the conditional the adversary knows `p == 0` implies `s == 0`. It also disallows array writes like `p[s] = 5` since the adversary can learn the value of `s` by seeing which element of the public array has changed. Note that accessing `s[p]` is safe because, despite knowing the index, an adversary cannot learn the value being accessed.

To prevent the length of a memory trace from revealing information, we require that loop guard expressions only involve public values (which is a standard restriction [43]). One can work around this problem by "padding out" loop iterations, e.g., by converting a loop like `while (slen > 0) { sarr[slen--]++; }` to be `plen = N; while (plen > 0) { if (plen <= slen) sarr[--plen]++; }` where `N` is a large, fixed constant. For similar reasons we also require that whether a function is called or returned from, and which function it is, may not depend on secret information (e.g., the call or return may not occur in a conditional whose guard involves secret information).

Compilation overview After source-language type checking, compilation proceeds in four stages—memory layout, translation, padding, and register allocation—after which the result is type checked using the $\mathcal{L}_T$ type system, to confirm that it is memory-trace oblivious.[5]

5.2 Memory bank allocation

The first stage of compilation allocates global variables to memory banks. Public variables are always stored in RAM, while secret variables will be allocated either to ERAM or ORAM. Two blocks in the scratchpad are reserved for secret and public variables, respectively, that will fit entirely within the block; these are essentially those that contain numbers, (pointers to) records, and small arrays. Such variables will be loaded into the scratchpad at the start of executing a program, and written back to memory at the end. The remaining scratchpad blocks are used for handling (large) arrays; the compiler will always use the same block for the same array. Public arrays are allocated in RAM, and secret arrays always indexed by public values are allocated in ERAM, and ORAM otherwise. The compiler initially assigns a distinct logical ORAM bank for each secret array, and allocates logical banks up to the hardware limit.

5.3 Basic compilation

The next stage is basic compilation (translation). Expressions are compiled by loading relevant variables/data into registers, performing the computation, and then storing back the result. Statements are compiled idiomatically to match the structure expected by the type rules in Figure 7 (with some work deferred to the padding stage).

Perhaps the most interesting part is handling variable accesses. Variables permanently resident in the scratchpad are loaded at the start of the program, and stored back at the end. Each read/write results in a **ldw**, to load a variable into a temporary register, and a **stw** to store back the result. Accesses to data (i.e., arrays) not permanently stored in the scratchpad will also require a **ldb** to load the relevant block into the scratchpad first and likewise a **stb** to store it back. A standard software cache, rather than a scratchpad, could eliminate repeated loads and stores of blocks from memory but could violate MTO. This is because a non-present block will induce memory traffic while a present block will not, and the presence/absence of traffic could be correlated with secret information. To avoid this, we have the compiler emit instructions that perform caching explicitly, using the scratchpad, with caching only enabled when in a public context, i.e., in a portion of code whose control flow does not depend on secret data. To support software-based caching, the compiler statically maps memory-resident data to particular scratchpad blocks, always loading the same data to the same block. Prior to doing so, and when safe, the compiler uses the **idb** instruction to check whether the relevant scratchpad block contains the memory block we want and loads directly from it, if so.

[5] This is essentially a kind of *translation validation* [38], which removes the compiler from the trusted computing base. We believe that well typed $\mathcal{L}_S$ programs yield well typed $\mathcal{L}_T$ programs, but leave a proof as future work.

Supporting functions requires handling calling contexts and local variables. We do this with two stacks, one in RAM and one in ERAM. Function calls are only permitted in a public context, which means that normal stack allocation and deallocation reveal no information, so no ORAM stack is needed. When a function is called, the current scratchpad variable blocks are pushed on the relevant stacks. At the start of a function, we load the blocks that hold the local variables. Local variables implementing ORAM arrays are stored by reference, with the variable pointing to the actual array stored in ORAM. This array is deallocated when its variable is popped from the stack, when the function returns (which like calls are allowed only in a public context).

The compiler is also responsible for emitting instructions that load code into the instruction scratchpad, as implicit instruction fetches could reveal information [33]. (To bootstrap, the first code block is loaded automatically.) At the moment, our compiler emits code that loads the entire program into the scratchpad at the start; we leave to future work support for on-the-fly instruction scratchpad use.

5.4 Padding and register allocation

Both branches of a secret conditional must produce the same trace. We ensure they do so by inserting extra instructions in one or both branches according to the solution to the *shortest common supersequence* problem [18]. When matching the two branches, we must account for the memory trace and instruction execution times. Only **ldb** and **stb** emit memory events; we discuss these shortly. While our formalism assumes each instruction takes unit time, the reality is different (cf. Table 2): times are deterministic, but non-uniform. For single-cycle operations (e.g., 64b ALU ops), we pad with **nop**s. For two-cycle **ldw** and **stw** instructions, we pad with two **nops**. For multiply and divide instructions, which take 70 cycles each, we could pad with 70 **nops** but this results in a large space overhead. As such, we pad both with the instruction $r0 \leftarrow r0 * r0$, where $r0$ is always 0. For conditionals, we pad the not-taken branch with two **nop**s, to account for the hardware-induced delay on the taken branch.

Padding for **stb** and **ldb** requires instructions that generate matching trace events. An access to ORAM is the simplest to pad, since the adversary cannot distinguish a read from a write. We can load any block (e.g., the first block of the ORAM) into a dedicated "dummy" scratchpad block, i.e. this block is used for loading and saving dummy memory blocks only.

For RAM and ERAM, the address being accessed is visible, so we need to make sure that the equivalent padding accesses the same address. To do this, the compiler should insert further instructions to compute the address. These instructions can be computed using the symbolic value: (1) if the symbolic value is a constant, then insert an assign instruction; (2) if the symbolic value is a binary operation of two symbolic values, then insert instructions to compute the two symbolic values respectively, and then another instruction to compute the binary operation; and (3) if the symbolic value is a memory value, then insert instructions to compute the offset first, and then insert a **ldw** instruction.

With instructions inserted to compute the address, we must emit either a load or a store depending on the instruction we are trying to match. For RAM, this instruction will always be a load because we perform padding in the H context, and the type system prevents writing to RAM. To mimic the **read**(l, k, sv) trace pattern, we first compute sv and then insert a **ldb** $k \leftarrow l[r]$ instruction where r stores the value for sv. To handle ERAM writes is challenging because we want the write to be a no-op but not appear to be so. To do this, we require the compiler to *always* follow an ERAM **ldb** with a **stb** back to the same address. In doing so, the compiler also prevents the padded instruction from overwriting a dirty scratchpad block.

At the conclusion of the padding stage we perform standard register allocation to fill in actual registers for the temporaries we have used to this point.

6. Hardware Implementation

We implement our deterministic processor by modifying Rocket, a single-issue, in-order, 6-stage pipelined CPU developed at UC Berkeley [41]. Rocket implements the RISC-V instruction set [51] and is comparable to an ARM Cortex A5 CPU. We modified the baseline processor to remove branch prediction logic (so that conditional branches are always not-taken) and to make each instruction execute in a fixed number of cycles. We describe the remaining changes below.

Instruction-set Extension We customize RISC-V to add a single data transfer instruction that implements **ldb** and **stb** from the formalism. We do this using a Data Transfer accelerator (Figure 2) that attaches to the processor's accelerator interface [50]. We also interface the Data Transfer accelerator with the x86-Linux host through Rocket's control register file so that it can load an `elf`-formatted binary into GhostRider's memory and reset its processor. Once this is done, the host performs processor control register writes to initiate transfers from the co-processor memory to the code ORAM for the code and data sections of the binary. The first code block of a program is loaded into the instruction scratchpad to begin execution; if subsequent instruction blocks are needed they must be loaded explicitly.

Scratchpads GhostRider has two scratchpads, one for code and one for data, each of which can hold eight 4KB blocks. The instruction scratchpad is implemented similar to an 8-way set-associative cache, where each way contains one block. The accelerator transfers one block at a time to a specified way in the instruction scratchpad. Once a block has been written, the valid and tag bits for that block are updated. The architecture does not implement the **idb** instruction from the formalism; instead, the compiler uses the first 8 bytes of every block to remember its address.

ORAM controller We implement ORAM by building on the Phantom ORAM controller [35] and implement an ORAM tree 13 levels deep (i.e., 2^{12} leaf buckets), with 4 blocks per bucket and an effective capacity of 64MB. ORAM controllers include an on-chip *stash* to temporarily buffer ORAM blocks before they are written out to memory. We set this stash to be 128 blocks. The Phantom design (and likewise, Ascend's [15–17]) treats the stash as a cache for ORAM lookups, which is safe when handling timing channels by controlling the memory access rate. GhostRider mitigates timing channels by having the compiler enforce MTO while assuming that events take the same time. As such, we modify Phantom's design to generate an access to a random leaf in case the requested block is found in the stash, to ensure uniform access times.

FPGA Implementation GhostRider is implemented on one of Convey HC-2ex's [10] four Xilinx Virtex-6 LX760 FPGAs. We measure hardware design size in terms of FPGA *slices* for logic and *Block RAMs* for on-chip memory. A slice comprises four 6-input, 2-output lookup tables (implementing configurable logic) and eight flip-flops (as storage elements) in addition to multiplexers, while each BRAM on Virtex-6 is either an 18Kb or 36Kb SRAM with up to two configurable read-write ports. The GhostRider prototype uses 47,357 such slices (39% of total) to implement both the CPU and the ORAM controller, and requires 685 of 1440 18Kb BRAMs (47.5%). Table 1 shows how these resources are broken up between the Rocket CPU and the ORAM controller, with the remaining resources being used by Convey HC-2ex's boilerplate logic to interface with the x86 core and DRAM. Note that this breakdown is a synthesis estimate before place and route.

Our prototype currently supports one data ORAM bank, one code ORAM bank, and one ERAM bank. We do not implement encryption (it is a small, fixed cost and uninteresting in terms of performance trends), and do not have separate DRAM; all public data is stored in ERAM when running on the hardware.

The Convey machine requires the hardware design to be run at 150 MHz while our ORAM controller prototype currently synthesizes to a maximum operating frequecy of 140MHz. Pending further optimization to meet 150 MHz timing, we run *both* the CPU and the ORAM controller in a 75 MHz clock domain, and use asynchronous FIFOs to connect the ORAM controller to the DDR DRAM controllers.

GhostRider simulator timing model In addition to demonstrating feasibility with our hardware prototype, we study the effect of GhostRider's compiler on alternate, more efficient ORAM configurations, e.g., Phantom at 150MHz [35] with two ORAM banks and a distinct (non-encrypting) DRAM bank. Hence we generate a timing model for both the modified processor and ORAM banks based on Phantom's hardware implementation [35], and incorporate the timing model into an ISA-level emulator for the RISC-V architecture; the model is shown in Table 2.

	Slices	BRAMs
Rocket	9287 (8.8%)	36 (10.5%)
ORAM	12845 (12.2%)	211 (61.5%)

Table 1. FPGA synthesis results on Convey HC-2ex.

Feature	**Latency** (# cycles)
64b ALU	1
Jump taken/not taken	3/1
64b Multiply/Divide	70/70
Load/Store from Scratchpad	2
DRAM (4kB access)	634
Encrypted RAM (4kB access)	662
ORAM 13 levels (4kB block)	4262

Table 2. Timing model for GhostRider simulator.

Name	Brief Description	Input Size (KB)
sum	Summing up all positive elements in an array	10^3
findmax	Find the max element in an array	10^3
heappush	insert an element into a min-heap	10^3
perm	computing a permutation executing $a[b[i]] = i$ for all i	10^3
histogram	compute the number of occurances of each last digit	10^3
dijkstra	Single-source shortest path	10^3
search	binary search algorithm	1.7×10^4
heappop	pop the minimal element from a min-heap	1.7×10^4

Table 3. Evaluated programs organized into programs with predictable, partially predictable, and data dependent memory access patterns (in order from top).

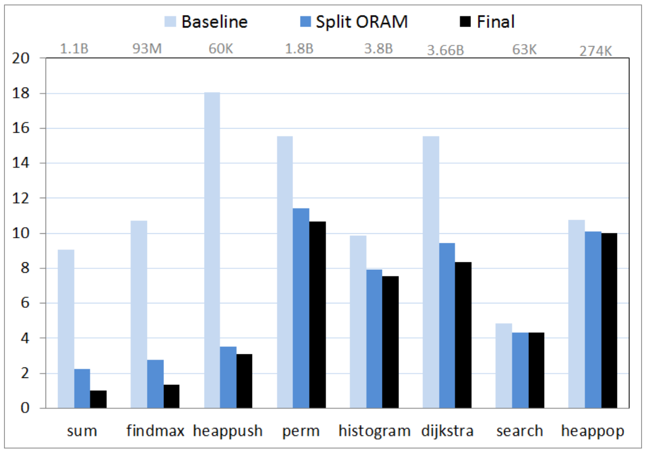
7. Empirical Evaluation

Programs Table 3 lists all the programs we use in our evaluation. These programs range from standard algorithms to data structures and include predictable, partially predictable, and predominantly irregular (data-driven) memory access patterns.

Execution time results We present measurements both for the simulator and for the actual FPGA hardware, starting with the former because the simulator allows us to evaluate the benefits from splitting memory into ERAM and ORAM banks v. additionally using a scratchpad. We also discuss the execution time results by categorizing them based on the regularity in the programs' access patterns.

`Non-secure`	Non-secure program: all variables in ERAM, no padding, and uses scratchpad.
`Baseline`	Secure baseline: all secret variables in a single ORAM, no scratchpad.
`Split ORAM`	Variables can be split across multiple ORAM banks, or placed in ERAM. Performs padding. No scratchpad.
`Final`	Scratchpad on top of `Split ORAM`.

(a) Legends.



(b) Slowdown in comparison with non-secure version.

Figure 8. Simulator-based execution time results.

Simulator-based results Figure 8 depicts the slowdown of various configurations relative to a non-secure configuration that simply stores data in ERAM and employs the scratchpad. Our non-secure baseline uses a scratchpad instead of a hardware cache in order to isolate the cost of MTO/ORAM. The secure `Baseline` configuration places all secret variables in a single ORAM, while `Split ORAM` employs the GhostRider optimization of using ERAM and multiple ORAM banks, and `Final` further adds the (secure) use of a scratchpad.

Three out of eight programs—`sum`, `findmax`, and `heap-push`—have a predictable access pattern and the secure program generated by GhostRider relies mainly on ERAM. Hence, each MTO program (`Final`) has almost no slowdown to 3.08× slowdown in comparison its non-secure counterpart (`Non-secure`), and correspondingly faster than `Baseline` by 5.85× to 9.03×.

For `perm`, `histogram`, and `dijkstra`, which have partially predictable and partially sensitive memory access patterns, our compiler attempts to place sensitive arrays inside both ERAM and ORAM and also favors splitting into several smaller ORAM banks without breaking MTO. As shown in Figure 8, for such programs, `Final` can achieve a 1.30× to 1.85× speedup over `Baseline` (with 7.56× to 10.68× slowdown compared to `Non-secure`, respectively).

For `search` and `heappop`, which have predominantly sensitive memory access patterns, the speedup of `Final`

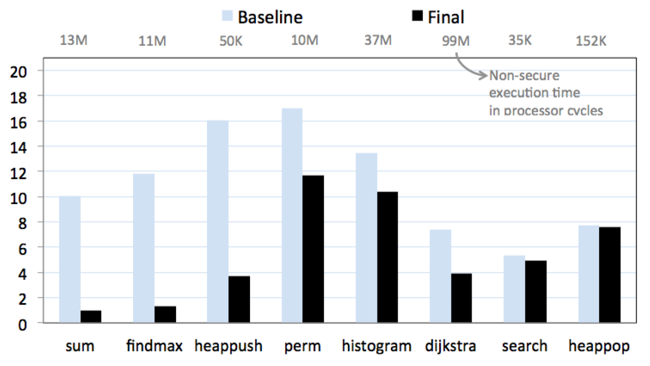


Figure 9. FPGA based execution time results: Slowdown of `Baseline` and `Final` versions compared to non-secure version of the program. Note that unlike Figure 8, `Final` uses only a single ORAM bank and conflates ERAM and DRAM (cf. Section 6).

over `Baseline` is not as significant, i.e. 1.07× and 1.12× respectively, and is due mostly to the usage of two ORAMs to store arrays instead of a single ORAM.

Examining the impact of the use of the scratchpad in the results, we can see that for the first six programs, `Final` reduces execution time compared to `Split ORAM` by a factor from 1.05× up to 2.23×.For `search` and `heappop`, the scratchpad provides no benefit because for these programs all data is allocated in ORAM, as array indices are secret (so the access pattern is sensitive), and our type system disallows caching of ORAM blocks. The reason is that the presence of the data in the cache could reveal something about the secret indices. A more sophisticated type system, or a relaxation of MTO, could do better; we plan to explore such improvements in future work.

FPGA-based results For the FPGA we run the same set of programs as in Table 3, but restrict the input size to be around 100 KB, due to limitations of our prototype. Speedups of `Final` over the secure `Baseline` follow a trend similar to the simulator, as shown in Figure 9. Regular programs have speedups in the range of 4.33× (for `heappush`) to 8.94× (for `findmax`). Partially regular programs like `perm` and `histogram` get a speedup of 1.46× and 1.3× respectively. Finally, irregular programs such as `search` and `heappop` see very little improvements (1.08× and 1.02× respectively).

Differences between the simulator and hardware numbers can be attributed to mulitple factors. First, the simulator imperfectly models the Convey memory system's latency, always assuming the worst case, and thus slowdowns compared to the non-secure baseline are often worse on the simulator (cf. `heappop` and `heappush`).

Second, the timing of certain hardware operations is different on the prototype and the simulator (where we consider the latter to be aspirational, per the end of Section 6). In particular, per Table 2, the simulator models access latency for ORAM as 4262 cycles and ERAM as 662 cycles, accounting for both reading data blocks from DRAM and moving the chosen 4KB block into the scratchpad BRAMs on the FPGA. On the hardware, ORAM and ERAM latencies are 5991 and 1312 cycles, respectively, measured using performance counters in the hardware design. The higher ERAM and ORAM access times reduce the slowdown on the simulator by amplifying the benefit of the scratchpad, which is used by the non-secure baseline, but not by the secure baseline (cf. `findmax` and `sum`).

Third, the benefit of using the scratchpad can differ depending on the input size. This effect is particularly pronounced for Dijkstra, where the ratio of secure to non-secure baseline execution is smaller for the hardware than for the simulator. The reason is that the hardware experiment uses a smaller input that fills only about 1/5 of a scratchpad block. Hence, in the non-secure baseline, the block is reloaded after relatively fewer accesses, resulting in a relatively greater number of block loads and thus bringing the performance of the non-secure program closer to that of the secure baseline.

Finally, note that the simulator's use of multiple ORAM banks, and DRAM with different timings, is a source of differences, but this effect is dwarfed by the other effects.

8. Related Work

Oblivious RAM was first proposed by Goldreich and Ostrovsky [21]. Since then, the community has made significant advances [20, 22, 23, 30, 53, 54] such that it evolved from a theoretical concept to real-life hardware prototypes [35, 40].

As an alternative to ORAM, *active RAM* capable of handling programmable logic may also be employed, such that memory addresses must be encrypted when transmitted over the memory bus. One candidate technology is the Hybrid Memory Cube [11]. Our compiler techniques are readily applicable to active RAM as well.

One line of past research has focused on designing customized data oblivious algorithms [7, 14, 24] such that they outperform generic ORAM simulation (i.e., placing all data in a single ORAM). This approach, however, provides point solutions to point problems, whereas our approach is general purpose and requires less human effort.

Secure type systems We build on ideas we previously proposed [33] for type-based enforcement of memory trace obliviousness. Our prior approach worked for a C-like language with only ORAM and RAM, whereas we consider a lower-level assembly language that additionally supports a scratchpad and ERAM; both extensions required nontrivial changes. Our full type system (see the full version [32]) also supports function calls and data structures. We also built a more full-featured compiler targeting a real architecture, rather than evaluating using a simpler simulation.

More generally, our work is the area of work on type-based enforcement of information-flow security [43], a topic sometimes applied to low level languages [3, 4, 6, 8, 28, 36, 39, 55]. Compared to these, our $\mathcal{L}_T$ type system is distinguished by its handling of a novel memory hierarchy. Our type system is essentially a kind of typed assembly language [37], but aims for information flow security rather than memory safety (and related properties). Finally, our compiler and type system are reminiscent of work that transforms programs to eliminate timing channels [2, 5, 12, 27] where inserted padding aims to equalize execution times. Ultimately, however, a program that is without timing leaks may still leak information via the memory trace.

Secure architectures HIDE [56] involves a compiler-assisted technique to obfuscate a processor's memory accesses but stops short of implementing provably secure oblivious RAMs. Shroud [34] is a storage system implementing ORAM using trusted co-processors (TPMs [1]) in datacenters; it uses complementary techniques to exploit parallelism in ORAM algorithms. Ascend [15–17] and Phantom [35] are two recent projects that have introduced adding oblivious computation to processors secure in the XOM model [42, 47, 48]—ORAM controllers presented in these works can be re-used in GhostRider. The key conceptual difference is that our MTO system uses the program's logic to optimize ORAM usage, and to account for the *length* of the memory trace, while Ascend and Phantom address timing and termination channels independent of the program's logic and ignore the length of the trace (hence may not terminate without leaking information).

9. Conclusion

We have presented the first complete memory trace oblivious system—GhostRider—comprising of a novel compiler, type system, and hardware architecture. The compiled programs not only provably satisfy memory trace obliviousness, but also exhibit up to nearly order-of-magnitude performance gains in comparison with placing all variables in a single ORAM bank. By enabling compiler analyses to target a joint ERAM-ORAM memory system, and by employing a compiler-controlled scratchpad, this work opens up several performance optimization opportunities in tuning bank configurations (size and access granularity) and, on a broader level, into co-designing data structures and algorithms for a heterogeneous yet oblivious memory hierarchy.

Acknowledgements We thank the anonymous reviewers, and our shepherd John Criswell, for their helpful feedback. This research was sponsored by NSF grants CNS-1111599 and CNS-1314857, the AFOSR under MURI award FA9550-09-1-0539, and the US Army Research laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] Trusted Platform Module (TPM) Summary. `http://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary`.

[2] J. Agat. Transforming out Timing Leaks. In *POPL*, pages 40–53, 2000.

[3] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *TLDI '05*, pages 103–112, 2005.

[4] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, 2010.

[5] G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, 2006.

[6] F. Bavera and E. Bonelli. Type-based information flow analysis for bytecode languages with variable object field policies. In *SAC*, pages 347–351, 2008.

[7] M. Blanton, A. Steele, and M. Aliasgar. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ASIACCS*, 2013.

[8] E. Bonelli, A. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In *CASSIS*, pages 37–56, 2006.

[9] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *ACM Cloud Computing Security Workshop (CCSW)*, pages 85–90, 2009.

[10] C. Computer. The convey HC2 architectural overview. `http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf`.

[11] H. Consortium. Hybrid memory cube. http://hybridmemorycube.org/.

[12] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE S & P*, pages 45–60, 2009.

[13] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *CSF*, pages 115–124, 2004.

[14] D. Eppstein, M. T. Goodrich, and R. Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[15] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[16] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, and S. Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, page 431, 2014.

[17] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

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

[19] T. Gilmont, J. didier Legat, and J. jacques Quisquater. Enhancing security in the memory management unit. In *EUROMICRO*, 1999.

[20] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[21] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[22] M. T. Goodrich and M. Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *ICALP*, pages 576–587, 2011.

[23] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[24] M. T. Goodrich, O. Ohrimenko, and R. Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[25] T. C. Group. Trusted computing group. `http://www.trustedcomputinggroup.org/`.

[26] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

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

[28] N. Kobayashi and K. Shirane. Type-based information flow analysis for low-level languages. In *APLAS*, 2002.

[29] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011.

[30] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme. In *SODA*, 2012.

[31] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *IEEE S & P*, 2003.

[32] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A hardware-software system for memory trace oblivious computation. Technical Report CS-TR-5041, University of Maryland, Department of Computer Science, Jan. 2015.

[33] C. Liu, M. Hicks, and E. Shi. Memory Trace Oblivious Program Execution. In *CSF*, 2013.

[34] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, 2013.

[35] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[36] R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In *ICTCS*, pages 360–374, 2005.

[37] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[38] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS*, 1998.

[39] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[40] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *ISCA*, 2013.

[41] riscv.org. Launching the Open-Source Rocket Chip Generator, Oct. 2014. `https://blog.riscv.org/2014/10/launching-the-open-source-rocket-chip-generator/`.

[42] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *MICRO*, pages 183 –196, 2007.

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

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

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

[46] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an Extremely Simple Oblivious RAM Protocol. IACR Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/280`.

[47] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171, 2003.

[48] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.*, 34(5):168–177, Nov. 2000.

[49] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms. In *ASIACCS*, May 2012.

[50] H. Vo, Y. Lee, A. Waterman, and K. Asanović. A Case for OS-Friendly Hardware Accelerators. In *WIVOSCA*, 2013.

[51] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[52] L. Whitney. Microsoft Urges Laws to Boost Trust in the Cloud. `http://news.cnet.com/8301-1009_3-10437844-83.html`.

[53] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.

[54] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

[55] S. A. Zdancewic. Programming Languages for Information Security. *PhD thesis*, 2002.

[56] X. Zhuang, T. Zhang, and S. Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News*, 32(5):72–84, Oct. 2004.