Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing

Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, Christopher W. Fletcher
University of Illinois at Urbana-Champaign
{jiyongy2, ljhsiun2, melhajj2, cwfletch}@illinois.edu

***Abstract*—Blocking microarchitectural (digital) side channels is one of the most pressing challenges in hardware security today. Recently, there has been a surge of effort that attempts to block these leakages by writing programs *data obliviously*. In this model, programs are written to avoid placing sensitive data-dependent pressure on shared resources. Despite recent efforts, however, running data oblivious programs on modern machines today is insecure and low performance. First, writing programs obliviously assumes certain instructions in today's ISAs will not leak privacy, whereas today's ISAs and hardware provide no such guarantees. Second, writing programs to avoid data-dependent behavior is inherently high performance overhead.**

This paper tackles both the security and performance aspects of this problem by proposing a *Data Oblivious ISA extension* (*OISA*). On the security side, we present ISA design principles to block microarchitectural side channels, and embody these ideas in a concrete ISA capable of safely executing existing data oblivious programs. On the performance side, we design the OISA with support for efficient memory oblivious computation, and with safety features that allow modern hardware optimizations, e.g., out-of-order speculative execution, to remain enabled in the common case.

We provide a complete hardware prototype of our ideas, built on top of the RISC-V out-of-order, speculative BOOM processor, and prove that the OISA can provide the advertised security through a formal analysis of an abstract BOOM-style machine. We evaluate area overhead of hardware mechanisms needed to support our prototype, and provide performance experiments showing how the OISA speeds up a variety of existing data oblivious codes (including "constant time" cryptography and memory oblivious data structures), in addition to improving their security and portability.

I. Introduction

With the rise of cloud computing and internet services, *digital or microarchitectural side channel attacks* [1] have emerged as a central privacy threat. These attacks exploit how victim and adversarial programs share hardware/virtual resources on shared remote servers (e.g., an amazon EC2 cloud). Simply by co-locating to the same platform, researchers have shown how attackers can learn victim program secrets through the victim's virtual memory accesses [2], [3], hardware memory accesses [4], [5], branch predictor usage [6], [7], arithmetic pipeline usage [8], [9], [10], speculative execution [11], [12] and more. Given the many avenues to launch an attack, it is paramount for researchers to explore holistic and efficient defensive strategies.

Recently, there has been a surge of work that attempts to block *all* digital side channels, on commercial machines, by writing and compiling programs in a *data oblivious* fashion (e.g., [13], [14], [15], [1], [16], [17], [18], [19], [20], [21], [22], [23], [24], [8], [25]). Data oblivious code, a.k.a. "constant time" or "running programs as circuits," blocks side channels by disallowing private data-dependent control flow. Figure 1 gives an example. Figure 1a leaks private information over microarchitectural side channels—namely, program execution time (the 'if-taken' case executes more instructions) and memory footprint (if `x` and `y` touch different lines in cache). To block these leakages, a data oblivious program will evaluate both sides of the branch as shown in Figure 1b. A ternary operator—e.g., implemented as the x86 `cmov` instruction or bitwise operations—chooses the correct final result (Figure 1b, Line 4). Since executing each side of the branch is independent of the secret, and the ternary operator does work independent of the secret, running the code data obliviously does not leak the secret.

```
1 x = 0, y = 64
2 if (secret)
3   x = y
4 z = Memory[x]
```

(a) Insecure code.

```
1 x = 0, y = 64
2 z = Memory[x]
3 tmp = Memory[y]
4 z = (secret) ? tmp : z
```

(b) Equivalent data oblivious code.

Fig. 1: Non-oblivious (1a) and equivalent data oblivious codes (1b). The word `secret` denotes private data.

A. Challenges

Despite the promise of data oblivious programs to block side channels, future progress faces two key challenges.

Security: Existing Instruction Set Architectures (ISAs) provide no guarantees that instructions used in data oblivious codes can block leakages over microarchitectural side channels. For example, if `cmov` (assumed to be a "secure" ternary operator in [1], [23], [17], [19]) was ever implemented as the microcode sequence `branch+mov`, the secret condition would leak through branch predictor state and whether hardware speculation results in a squash. Being ISA-invisible, these changes can occur at any time. Case in point, Intel has stated that `cmov`'s behavior w.r.t. speculation may change in future processors ([26], Section 3.2).

Beyond `cmov`, the larger problem is that commercial ISAs such as x86 give engineers significant rope to perform *data-dependent* optimizations during program execution. For example, it is well known that arithmetic units can sometimes take data-dependent time [8], [9]. We provide a comprehensive background on related vulnerabilities in Section III-B. Any of these software-invisible optimizations can undermine the security of prior and future work that attempts to write data oblivious programs.

Performance: Data oblivious codes can incur large per-

formance overheads. The reason, once again, is that data obliviousness does not have ISA-level support. As a result, programmers are forced to use only the simplest instructions to achieve data oblivious execution, out of fear that other instructions will leak privacy. For example, data oblivious codes must make two memory accesses in Figure 1b out of fear that a single access will reveal the address through the processor cache, or other, side channel. This overhead scales with deeper data-dependent control flow and larger data sizes.

B. This Paper

In this paper, we tackle both the security and performance aspects of this problem by developing a novel type of ISA extension which we call a *Data Oblivious ISA* extension (*OISA*). To our knowledge, this represents the first foundation for writing and executing secure, portable and performant data oblivious code on commercial-class (out-of-order, speculative) processors. To this end we make the following contributions:

1.) Design principles for OISA design. Our key idea is to explicitly specify security guarantees at the ISA level, while decoupling those guarantees from the implementation details of a particular processor. Our abstraction is two parts. First, the ISA denotes *data* to be Public or Confidential. Second, the ISA denotes each *instruction operand* to be Safe or Unsafe. If an instruction with Safe operands consumes Confidential data, processor implementations (*microarchitectures*) must hide attacker-observable data-dependent side effects stemming from that instruction's execution. If an instruction with Unsafe operands consumes Confidential data, the hardware must throw an exception before data-dependent behavior can occur. If either type of operand consumes Public data, the hardware is free to apply optimizations to improve performance.

Importantly, how protecting Safe operands is implemented is left to the hardware designer, who can devise efficient protections depending on each microarchitecture (e.g., by breaking the instruction into simpler data oblivious instructions [8] or using hardware partitioning [27] or using cryptographic techniques [23]). In all cases, the programmer works with a simple, portable guarantee.

2.) Design of a concrete OISA. With these principles, we define a set of instructions that can serve as the foundation for the rich line of ongoing work in data oblivious programming [14], [13], [17], [18], [19], [20], [24], [23], [22], [1], [21], [8], [25]. Beyond Turing completeness and security, we also want to reduce the performance overhead common with data oblivious code. To that end, we provide additional instructions that implement efficient *memory oblivious* computation [23], [21] (featuring loads/stores with private addresses). Given the principles above, this extension is conceptually simple: instead of emulating memory obliviousness with dummy memory operations (Figure 1b), we designate a new load instruction whose address operand is *Safe*, which gives hardware designers the ability to build secure and efficient implementations, e.g., using partitioning, for that specific operation.

Load instructions with Safe addresses are just one example of how to accelerate data oblivious code with an OISA. A key insight is that many data oblivious codes share common kernels (e.g., sorting [23], [19], [28]) that often become performance bottlenecks due to the cost of implementing them obliviously with simple instructions. By encapsulating those larger operations into new instructions with Safe operands, an OISA can achieve constant factor or even asymptotic performance improvements. For example, a sort implemented obliviously with simple instructions may cost $O(n*\log^2 n)$ time if implemented as a bitonic sort [29]. On the other hand, if sort is specified as an instruction in the OISA, an implementation based on hardware partitioning can achieve $O(n*\log n)$ time if implemented as a constant time merge sort.

3.) Hardware prototype on an out-of-order, speculative processor. To show that our ideas are practical, we prototype all hardware changes needed to support our ISA on top of the RISC-V BOOM processor (for "Berkeley Out-of-Order Machine") [30]. BOOM is the most sophisticated open RISC-V processor, featuring modern performance optimizations such as speculative and out-of-order execution, and is similar to commercial machines that run data oblivious code today.[1]

4.) Formal analysis: non-interference on out-of-order, speculative execution-class processors. In parallel to our hardware prototype, we develop a formal analysis that models an abstract BOOM-class processor (out-of-order, speculative, superscalar), and describe how to map the abstract BOOM to our concrete BOOM prototype. A key insight enabling (and simplifying) this analysis is that with the OISA, speculative execution need not be handled as a new case. In particular, our OISA requires the hardware to perform local permissions checks on each instruction as it executes, and these checks need not be aware of whether each instruction is speculative, executed out-of-order, etc. Through this formalism, we prove that the OISA provides a basis to satisfy strong security definitions such as non-interference [34] on advanced machines. Importantly, we achieve this result *while* allowing high performance hardware optimizations (e.g., out-of-order, speculative execution) to remain enabled in the common case and *without* ever requiring hardware flushes to structures such as the cache or branch predictors [35], [36].

5.) Evaluation. We evaluate our proposal in terms of hardware area and performance over a range of existing data oblivious programs (including linear algebra, data structures, and graph traversal). Area-wise, our proposal takes $< 5\%$ the area of the unmodified BOOM processor. Performance-wise, our ISA and hardware implementation provides an $8.8\times/1.7\times$ speedup on small/large data sets, respectively, relative to data oblivious code running on commodity machines (and with the security and portability benefits stated before). We also show case studies, where our ISA speeds up constant time AES [37], [38] by $4.4\times$ and the memory oblivious ZeroTrace [23] library by $4.6\times$ to several orders of magnitude, depending on parameters.

We have open-sourced our prototype design on the RISC-V BOOM processor at https://github.com/cwfletcher/oisa.

[1] We note that prior work [31], [32], [33] requires the use of discrete co-processors with simple microarchitecture. To match modern cloud deployments, our goal is to support *concurrent* execution of many processes on advanced microarchitectures.

II. Background and Threat Model

A. Hardware Terminology

1) Out-of-order execution: Modern commercial processors such as the RISC-V BOOM [30] dynamically schedule and execute data-independent instructions in parallel and out of program order to improve performance. Instructions are *fetched* and *issued* (enter the scheduling system) in *program order*, *execute* (perform their operations and produce their results) possibly out of program order, and finally *retire* (make their operation externally visible by irrevocably modifying the architected system state) in program order.

2) Speculative execution: Speculative execution improves performance by executing instructions whose validity is uncertain instead of waiting to determine their validity. If such a speculative instruction turns out to be valid, it is eventually retired; otherwise, it is *squashed* and the processor's state is rolled back to a valid state. (As a byproduct, all following instructions also get squashed.) That is, a squash causes a large pipeline disturbance. There are multiple ways an instruction stream can be speculative—e.g., due to branches, memory accesses [39], or even arithmetic instructions [40]—discussed further in Section III-B.

More details on commodity out-of-order pipeline designs (BOOM as an example) are given in Section V-A.

B. Threat Model

We consider the setting where a victim program runs on a shared machine in the presence of adversarial software. The adversary's goal is to learn private data in the victim program through digital side channels. For example, private inputs contributed by another party or secret program state (e.g., a cryptographic key). The program itself is considered public. We trust the processor hardware and that the victim program is correctly using the OISA.

We defend against two classes of adversary: supervisor-level (Ring-0) or user-level (Ring-3) software. In both cases, we strive to block digital side channels that could be exploited by the standard Intel SGX adversary used in prior work on data oblivious programming [19], [1], [23], [24], [16], [17], [18], [21]. This adversary is supervisor-level software that controls when victim threads run, and therefore can monitor/influence the victim's hardware resource utilization (e.g., monitor/prime the cache/branch predictors [4], [12], [36]) at near-perfect resolution (e.g., via [41], [2], [3]). By extension, this adversary can monitor the victim's termination time, and determine when a precise exception [1], [42] or system call [43] occurs. We don't make assumptions on where the victim runs relative to adversarial code (e.g., as an adjacent SMT context, adjacent core, etc.). If the adversary is actually user-level software, our threat model is strictly conservative.[2]

In the case of a supervisor-level adversary, we assume the victim is running within a virtual shielding system, such as an SGX enclave [45], [46], to prevent direct inspection/tampering on victim data. The OISA is orthogonal to which virtual shielding system is used, in the sense that shielded programs can execute oblivious instructions regardless of the exact shielding system implementation. We will therefore only discuss the OISA, independent of the shielding system, for the rest of the paper.

Non-goals. Physical side channels (e.g., power [47] or EM [48]) are out of scope. Similar to previous works on data oblivious programming, we also do not consider integrity of computation. Integrity relies on orthogonal mechanisms, e.g., traditional or SGX-augmented process/memory isolation.

III. Data Oblivious Execution

We now give background on data oblivious execution and give examples for where prior work on commercial ISAs (e.g., x86) and modern machines (e.g., speculative, out-of-order) is vulnerable to attack.

A. Security Definition

Data oblivious execution satisfies computational indistinguishably[3] of program traces, once the trace is projected by an appropriate observability function.

Definition III.1. *(Confidential input privacy). Given a program λ with Public (non-sensitive) input x and Confidential (sensitive) input y, $\mathsf{O}(\mu Arch(\lambda(x,y))) = X = \{X_0, X_1, \ldots, X_M\}$ represents the program's observable execution trace (projected through function O) when running on a processor $\mu Arch$. What information is contained in each X_t (for each time step t) depends on the observability function O. W.l.o.g. we will treat x and y as fixed-size arrays, thus λ can accept an arbitrary number of Public and Confidential inputs. Privacy for the Confidential inputs then requires:*

$$\forall x \in Data_P,\ \forall y, y' \in Data_C : \\ \mathsf{O}(\mu Arch(\lambda(x,y))) \simeq \mathsf{O}(\mu Arch(\lambda(x,y')))$$

where $\simeq$ denotes computational indistinguishability, and $Data_P$ and $Data_C$ denote the space of Public and Confidential inputs, respectively.

We denote Definition III.1 parameterized by an observability function O (which models the adversary's view) and a specific microarchitecture $\mu Arch$ as $Oblivious[\mathsf{O}, \mu Arch]$, dropping $\mu Arch$ when it is clear which microarchitecture we are referring to.

Existing data oblivious programs written for commodity machines demand a rich observability function that reveals fine-grain details about processor state [14], [13], [17], [18], [19], [20], [24], [23], [22], [1], [21], [25]. The reason is that machines today are shared, and adversaries from Section II-B can monitor internal activity such as caches and pipeline behavior. It is therefore useful to define the most conservative observability function that could apply to adversaries from Section II-B:

Definition III.2. *(*BitCycle *observability: Security labels at **bit**-level spatial granularity, **cycle**-level temporal granularity). Let $S_t = \{0,1\}^N$ denote the processor state during clock cycle t, where state includes all on-chip storage (e.g., flip-flops,*

[2]Note that even user-level adversaries have been shown to be surprisingly powerful in their ability to monitor digital side channels [44].

[3]Here, computational indistinguishability (adopted from the Oblivious RAM literature [49]) is synonymous with computational non-interference [50], and the definition can be easily changed to require strict non-interference [34] if the program does not require computational assumptions.

SRAM). S_t^i denotes the value of the i-th bit in cycle t. Given a program execution $\lambda(x,y)$, $\mathsf{BitCycle}(\mu Arch(\lambda(x,y))) = X = \{X_0, X_1, \ldots, X_M\}$ where $X_t = \{0,1\}^N$ and $X_t^i = 1$ indicates S_t^i contains an explicit flow[4] of Confidential data in cycle t ($X_t^i = 0$ otherwise).

The idea with this definition is to model an adversary that cannot directly inspect the logic value in each memory cell, but can measure resource usage (i.e., when each cell is or is not being used for another program's execution). For example, writing data d to the processor cache at address a in cycle t sets bits in X_t, corresponding to cache memory cells at a, if either d or a were computed based on Confidential data. BitCycle, specifically, implies the adversary can monitor every possible hardware resource pressure (e.g., flip-flop level pipeline utilization, cache footprint, etc.) every cycle. Courser models are also possible depending on the anticipated adversary or existing defense mechanisms (e.g., [27]). This paper's goal is to provide a basis for programs to achieve *Oblivious*[BitCycle], or a courser model, on advanced commercial-class machines.

B. Security Issues in Existing Data Oblivious Code

Existing data oblivious codes are written extremely conservatively to remove code constructs that blatently violate *Oblivious*[BitCycle]. For example, prior works rely solely on a carefully chosen subset of arithmetic operations (e.g., bitwise operations), conditional moves, branches with data-independent outcomes, jumps with public destinations, and memory instructions with data-independent addresses [13], [14], [15], [1], [16], [17], [18], [19], [20], [21], [22], [23], [24], [8], [25].

It is important to understand when this isn't sufficient for security. To that end, we now detail 11 possible attack vectors on today's data oblivious code. Importantly, we do not list many popular attacks (e.g., prime+probe in the cache [4]) as these are defeated by writing programs in the style described above. Yet, attacks can still occur because the hardware can apply invisible optimizations to undermine software-level transformations. In the following, we describe attack vectors known to be implemented today, and also proposals whose implementation status is unknown. However, importantly, each optimization could be implemented at any time, breaking existing codes.

Vectors 1, 2, 3: branch, jump, memory speculation: While transient execution attacks [12], [11] are known to impact general purpose code, their impact on data oblivious code has not been adequately studied. We make an important observation that data oblivious code security is undermined even by 'honest' speculative execution. By 'honest', we mean the speculation is not intentionally being controlled in a malicious way, e.g., as in [12]. The root problem is that modern ISAs have limited resources (e.g., ISA-level registers) and executing unintentional instructions can cause secrets stored in *aliased resources* to be exposed accidentally.

[4]Formally: Let each memory cell S^i take two inputs: data (in^i) and write enable (we^i) where both are functions (combinational logic) taking a subset of bits in S as input. For time $t = 0$: S_0 (i.e., at $t = 0$) is initialized with starting program state, $X_0^i = 1$ iff S_0^i is Confidential data. For time $t > 0$: $X_t^i = 1$ if (a) we^i outputs 0 in cycle t and $X_{t-1}^i = 1$ or (b) we^i outputs 1 in cycle t and $X_{t-1}^j = 1$ for some j in the inputs to S^i (in^i or we^i). We note that implicit flows [51] are accounted for once BitCycle is applied to Definition III.1.

Consider a toy example for data oblivious decryption, exploiting conditional branch misprediction (denoted Vector 1):

```
1 for (i = 0; i < NUM_ROUNDS; i++)
2   state = OblDecRound(state, rkey[i])
3 declassify(state)
```

A legal data oblivious code can implement decryption round logic data obliviously, with the round keys `rkey` considered Confidential (Definition III.1), and wrap the round in a data-independent branch to reduce code footprint. Once decryption is complete, the program may use the plaintext in a non-oblivious way, e.g., by using it as an address to lookup a record in cache (denoted `declassify(state)`). Such a non-oblivious operation can reveal information related to the *decryption key* on a speculative machine. Specifically, if the branch mispredicts "not taken" (e.g., while the predictor is training), `state` is prematurely exposed before all rounds complete, allowing an attacker to perform cryptanalysis on encryption round intermediate state.

Removing branches or disabling branch speculation is not sufficient to fix this issue, as other forms of speculation (e.g., unconditional branches/jumps, memory disambiguation [11]—denoted Vectors 2 and 3) cause similar issues on legal data oblivious code.

Vectors 4, 5: sub-address optimizations: Numerous data oblivious codes, e.g., "constant time" cryptography [52], [53], make an assumption that modulating certain bits in a memory address (e.g., the bits indicating offset within a cache line) does not create observable behaviors. This assumption doesn't hold on some microarchitectures due to hardware optimizations such as speculative store forwarding (Vector 4) and cache banking (Vector 5), and attacks exploiting these features have been shown to lead to full cryptographic breaks [54], [55].

Vector 6: input-dependent arithmetic: It is well known that complex arithmetic operations (e.g., multiply/divide, floating point square root) exhibit observable data-dependent timing based on their operands [9], [8]. While prior work can mitigate these threats by re-writing complex arithmetic using bitwise operations, this can incur over an order of magnitude performance overhead depending on the operation [8].

Vector 7: microcode: Even simple instructions may be decomposed into simpler instructions, called micro-ops, before being executed. In some cases, micro-op conversion can create data-dependent behavior. For example, `cmov` (which implements conditionals based on Confidential values [1], [23], [17], [19]) can be broken into a `branch+mov`. There is evidence to suggest that this transformation will be applied in future Intel processors ([26], Section 3.2). This breaks privacy: the branch direction will be speculatively guessed and whether a misprediction occurs changes program timing due to the squash (Section II-A).

Vectors 8, 9, 10, 11: data-based compression, data-based speculation, silent stores: Finally, there are a number of proposals whose implementation status on commercial machines is unknown. In register file [56] and cache [57] compression (analogous to OS-level page de-duplication [58]), register file and cache pressure is a function of program data (Vectors 8 and 9, respectively). Value prediction [40] (Vector 10) speculates

on the result of a memory load or long-running arithmetic operation, causing a squash if the prediction is incorrect (Section II-A). Finally, silent stores [59] (Vector 11) remove redundant store operations (impacting cache pressure) when the hardware detects the memory already contains the same value at the same address. What all of the above have in common is that they are *program data*-centric optimizations that don't discriminate between Public and Confidential data. Thus, they can undermine any data oblivious code written in any style.

Takeaway: Not only is writing data oblivious code difficult, it is fraught with danger due to subtle ISA-invisible optimizations such as those given above. Our proposed OISA gives hardware the visibility it needs to decide when and when not to apply leaky performance optimizations (such as those above) and enables richer hardware support for data oblivious code to speedup core operations such as oblivious memory.

IV. Data Oblivious ISAs

We now describe data oblivious ISA (OISA) design principles and give an example concrete OISA that we will later implement on top of the RISC-V BOOM.

A. Design Principles

We had two primary goals in designing an OISA. First, the ISA should expose security guarantees in a microarchitecture-independent way. A single ISA may be embodied in many different microarchitectures (within and across processor generations), each with different organizations and optimizations. It isn't reasonable to ask software to reason about each microarchitecture: a developer who writes a data oblivious code correctly once should have confidence that security will hold on each microarchitecture. Second, the ISA should not preclude modern hardware performance techniques, except when those techniques have a chance to leak privacy. Specifically, we want to be compatible with wide (multiple instructions fetched per cycle), speculative, out-of-order commercial-class machines, e.g., those described in Section II-A, and also point optimizations (e.g., banked caches, data-dependent arithmetic; c.f. Section III-B) that, left unchecked, cause security problems.

To achieve these goals, an OISA adds a two-part abstraction to an existing ISA, which we summarize here. First, the ISA denotes *data* to be Public or Confidential. Second, the ISA denotes each *instruction operand* to be Safe or Unsafe. If an instruction with Safe operands consumes Confidential data, processor implementations (*microarchitectures*) must hide attacker-observable data-dependent side effects stemming from that instruction's execution. If an instruction with Unsafe operands consumes Confidential data, the hardware must throw an exception before data-dependent behavior can occur. If either type of operand consumes Public data, the hardware is free to apply optimizations to improve performance.

We now describe more detail for how hardware can enforce both of these components.

1) Dynamic tracking for Confidential (sensitive) data: We use hardware-based dynamic information flow tracking techniques (DIFT, similar to [60], [61]) to track how Confidential data propagates through the processor as the program executes. Conceptually, all data in the processor is *labeled* Confidential/Public at some granularity (e.g., word-level).[5] This gives hardware the ability to decide when to apply optimizations to data in use (e.g., attack Vectors 6-7, 10-11; c.f. Section III-B) and at rest (e.g., Vectors 8-9).

Prior work does not specify precise rules for when data labeled Confidential can be processed relative to when its label is resolved. A conservative strategy is to require all {data, label} state to correspond to program order, which would preclude speculative, out-of-order execution. A more aggressive strategy is to allow speculation, and to further allow data to be used before its label is resolved.[6] Based on our use of DIFT, it will be clear the latter approach is not secure. Instead, we adopt (and prove secure in Section VI) a middle ground which we call *coherent labels*.

Rule IV.1. *(Coherent labels) When reading an operand, its label must be resolved with respect to the dynamic sequence of speculative/non-speculative instructions (which does not necessarily follow program order) that have executed so far to generate that operand.*

A simple implementation that satisfies Rule IV.1 is to physically extend each data word with a label bit, which allows normal processor dependency tracking to ensure labels are resolved on time. We use this strategy for our implementation in Section V.

2) Instruction operand-level security specifications: In an OISA, instruction definitions specify, for each operand, whether that operand can accept Public or both Public/Confidential data. We call the former an *Unsafe* operand and the latter a *Safe* operand. Once specified, the hardware designer must handle the following cases.

Rule IV.2. *(Confidential → Safe) When Confidential data is sent to a Safe operand: the hardware designer must add mechanisms to enforce Definition III.1, for a specified observability function, despite that instruction's execution. For example, by disabling performance optimizations, scrubbing side effects and masking exceptions that occur as a function of Confidential operands.*

Rule IV.3. *(Confidential → Unsafe) When Confidential data is presented to an Unsafe operand: the hardware must stop (squash) that instruction's execution as soon as the label is resolved. This event is called a label violation* #LV*. Due to Rule IV.1,* #LV *will be signaled immediately after register/memory read, and before the execute stage begins. If the violating instruction is the next instruction to retire (i.e., is non-speculative), terminate the program. This event is called a label fault* #LF*.*

That is, Rule IV.3 is similar to rules that handle badly typed programs, extended to speculative execution. Label violations (#LV) are caused by transient conditions, e.g., imperfect prediction (Section III-B, Vector 1), and are correctable. Label faults (#LF) indicate a program bug or illegal typing. Fixing bugs is outside of our scope, so we will focus on #LV.

[5] 'Public' and 'Confidential' semantics are equivalent to the lattice $\{L,H\}$ ({low, high} security) where $L \sqsubseteq H$ [62], [63].

[6] For example, [61] proposes storing labels in the page table. If the processor supports speculative store-forwarding [54] (Vector 4), data will be used before the label lookup completes.

An important question is whether #LV creates a side channel based on *when* it is triggered. We prove in Section VI-B that it does not, and further prove that #LV signals enable the OISA to block multiple additional attacks (Vectors 1-5; c.f. Section III-B), e.g., speculation that can reveal Confidential data, on top of the vectors blocked from Section IV-A1. Finally, Public data is handled as:

Rule IV.4. *(Public → Safe/Unsafe) When Public data is sent to Safe or Unsafe operands, no special treatment is needed and execution can proceed without protection.*

As the above definitions apply at operand granularity, the OISA permits optimizations that are functions of individual operands. For example, zero-skip multiply can be enabled if a Public operand is 0, regardless of whether other operands are Confidential.

Specifying each instruction operand as Safe/Unsafe at the ISA level is a key design feature, and provides significant flexibility to both the ISA and hardware designer while simplifying programmer-level reasoning about security. At the ISA level, an ISA designer can decide which instructions are sufficiently important to warrant Safe operands. These choices should be made carefully: On one hand, Safe operands impose a burden on hardware designers as the processor must support mechanisms to uphold Definition III.1 for those operands. On the other hand, Safe operands do not specify an implementation strategy. Hardware designers can implement a given operation using simpler data oblivious instructions (e.g., [8]), hardware partitioning (e.g., [27]) or cryptographic techniques (e.g., [23])—depending on what is efficient given public parameters and the specific microarchitecture. In either case, programmers work with a simple guarantee: Confidential values will not be at risk when consumed by Safe operands, and dynamic execution will be terminated when violations to this policy are detected.

B. Concrete OISA Specification

Using the principles from the previous section, we now present a concrete OISA that we will implement on top of the RISC-V BOOM processor. Figure 2 shows data oblivious instruction encodings, supported instruction types, and the Safe/Unsafe characteristics for each operand (Section IV-A2).

1) Label propagation: Our ISA requires word-granularity labels, tracked in the register file and memory. In most cases, label update logic follows standard taint tracking rules, given the 2-level security lattice {Public, Confidential} [62], as shown in Figure 2. When the result is fully determined by Public operands, regardless of other operands (e.g., zero-skip multiply), the result label is set to Public (as done in GLIFT [65], but not shown in Figure 2 for simplicity).

2) Label declassification: Declassification—downgrading data marked Confidential to Public—is a rare but necessary task needed to, e.g., return results. Our ISA supports a single serializing declassification instruction called ounseal. Serializing instructions are not executed until all older in-flight instructions retire. This is necessary for security: declassification is the only mechanism to demote Confidential to Public, and this action under malicious speculative execution could be used to bypass label checking.

3) Instruction set: Our ISA supports the following instruction types, which we chose to maximize compatibility with existing data oblivious codes and minimize hardware changes. First, all RISC-V integer and floating point arithmetic with Safe operands. This means programmers can implement floating point directly, without invoking bitwise libraries [8]. Second, random number generation, as many randomized data oblivious codes require private random numbers (e.g., [66], [67], [68], [23], [21]). Third, a `cmov`-style ternary/conditional move operator with a Safe predicate for implementing conditionals, and branches/jumps with Unsafe operands to reduce code footprint. Fourth, load/store operations (orld and orst) with Unsafe address operands.

Lastly, we support a second flavor of load/stores (with Safe address operands) which can be used to implement oblivious memory using Confidential addresses (Section IV-B6).

4) Mixing in non-oblivious instructions: Oftentimes, only a small program region should be made data oblivious (e.g., the inner branch in modular exponentiation) to prevent unnecessary performance overheads. To support these situations, we support mixing data oblivious instructions with instructions from the original ISA. All operands for all original instructions are considered Unsafe. All data oblivious instructions are encoded on top of the normal RISC-V ISA by modifying existing instruction fields (e.g., the opcode and func [64]).

5) Putting it all together: To summarize the section, we show a version of Figure 1b written using our OISA in Figure 3a. The programmer need only specify what data is Confidential via oseal. The ISA and hardware will prevent `%x3` from being processed by subsequent speculative/non-speculative Unsafe operands. For example, specifying `%x3` as an address to a speculative/non-speculative orld triggers a #LV/#LF, respectively.

6) Oblivious memory extension: A common bottleneck in existing data oblivious code is the inability to use Confidential data as memory addresses [23], [1], [27], [21]. For example, Figure 3a needed to execute two orld instructions. More generally, looking up an array with a Confidential address requires a memory scan.

To accelerate these operations, our OISA exposes two new instructions ocld and ocst, which are analogous to orld/orst (Section IV-B3) except with Safe address operands, and a new variant of CPUID ocpuid which returns a microarchitecture-specific constant OSZ ("oblivious memory partition size").

Each microarchitecture is responsible for providing OSZ bytes of "fast" oblivious storage, called the *oblivious memory partition* (OMP), which only ocld and ocst can read/write. This storage can be used to speedup data oblivious code. For example, if `x` and `y` in Figure 1b both fall within the OMP, then Figure 3a can be rewritten as Figure 3b (saving a memory access).

How much storage is provided (the value of OSZ) and how that storage is implemented—e.g., a dedicated scratchpad, flexible cache partition, etc.—is left to hardware designers and can be decided on an implementation-by-implementation basis. (Our prototype in Section V-B uses ways in a cache.) We note that the hardware constrains addresses sent to ocld/ocst to fall within bounds 0 to OSZ-1.

Base Data Oblivious ISA:	Operand label constraints (S = Safe, U = Unsafe)		Instruction functionality	Label propagation	Notation (assembly)
Arithmetic (R-type)	rs2 (S)	rs1 (S)	**R**[rd] <- **R**[rs1] op **R**[rs2]	**Lr**[rd] <- **Lr**[rs1] | **Lr**[rs2]	
Arithmetic (I-type)		rs1 (S)	**R**[rd] <- **R**[rs1] op ext(imm)	**Lr**[rd] <- **Lr**[rs1]	
Declassify (I-type) (**serializing**)		rs1 (S)	**R**[rd] <- **R**[rs1]	**Lr**[rd] <- 0	ounseal %rd, %rs1
Classify (I-type)		rs1 (S)	**R**[rd] <- **R**[rs1]	**Lr**[rd] <- 1	oseal %rd, %rs1
Conditional move (R-type)	rs2 (S)	rs1 (S)	**R**[rd] <- (**R**[rs1]) ? **R**[rs2] : **R**[rd]	**Lr**[rd] <- **Lr**[rs1] | **Lr**[rs2] | **Lr**[rd]	ocmov %rd, %rs1, %rs2
Branch (B-type)	rs2 (U)	rs1 (U)	if (**R**[rs1] op **R**[rs2]) PC = PC + imm	-	
Jump register (I-type)		rs2 (U)	**R**[rd] = PC + 4; PC = PC + imm	**Lr**[rd] <- 0	
RNG (J-type)			**R**[rd] <- rand()	**Lr**[rd] <- 1	orng %rd
Load (I-type)		rs1 (U)	**R**[rd] <- **M**[**R**[rs1]+ext(imm)]	**Lr**[rd] <- **Lm**[**R**[rs1]+ext(imm)]	orld %rd, imm(%rs1)
Store (S-type)	rs2 (S)	rs1 (U)	**M**[**R**[rs1] + ext(imm)] <- **R**[rs2]	**Lm**[**R**[rs1]+ext(imm)] <- **Lr**[rs2]	orst %rs2, imm(%rs1)
Oblivious Memory extension:			let addr := **R**[rs1]+ext(imm) % OSZ		
Oblivious Load (I-type)		rs1 (S)	**R**[rd] <- **M**[addr]	**Lr**[rd] <- 1	ocld %rd, imm(%rs1)
Oblivious Store (I-type)	rs2 (S)	rs1 (S)	**M**[addr] <- **R**[rs2]	-	ocst %rs2, imm(%rs1)
CPUID (J-type)			**R**[rd] <- OSZ	-	ocpuid %rd

Fig. 2: Data Oblivious ISA extension. **R/Lr**, **M/Lm** denote register file data/labels, memory data/labels, respectively. The label Public is denoted logic 0, Confidential logic 1. rs1 and rs2 denote operand registers in RISC-V instructions while rd denotes destination register. R, I, B, J, S-type refers to standard RISC-V instruction formats [64]. ext extends the immediate to the word width. If assembly notation is unspecified, it follows RISC-V with an 'o' prefix (e.g., add becomes oadd). OSZ refers to the microarchitecture-specific oblivious memory partition size (Section IV-B6). Note that all locations in the partition are implicitly marked Confidential at all times.

```
1 oaddi %x1, %x0, 0
2 oaddi %x2, %x0, 64
3 oseal %x3, secret
4 orld %x1, 0(%x1) //Mem
5 orld %x2, 0(%x2) //Mem
6 ocmov %x1, %x3, %x2
```

(a) Data obl. Fig. 1b.

```
1 oaddi %x1, %x0, 0
2 oaddi %x2, %x0, 64
3 oseal %x3, secret
4 ocmov %x1, %x3, %x2
5 ocld %x1, 0(%x1) //Mem
```

(b) Data obl. Fig. 1b w/ OMP.

Fig. 3: Data oblivious code, using the OISA, implementing Figure 1b. The word `secret` denotes Confidential data. `%x...` are RISC-V general purpose registers. `%x0` is a RISC-V idiom for constant 0.

To make data oblivious code portable across machines (each of which can specify a different OSZ), we provide the following software/programmer-level functions:

- Unsafe OblObj * obl_alloc(Unsafe int size)
- void obl_free(Unsafe OblObj * o)
- Safe int obl_read(Unsafe OblObj * o, Safe int addr)
- void obl_write(Unsafe OblObj * o, Safe int addr, Safe int data)

Safe/Unsafe qualifiers are implied based on how these functions are implemented. That is, size must be Public. obl_alloc/free dynamically allocate/free an oblivious memory object OblObj which exposes type, base and bound fields. type = {OMP, ORAM, SCAN} and is determined by obl_alloc under the hood using the following rules:

1) If the new object will completely fit into the OMP, based on the size argument, previous allocations, and OSZ: set type = OMP.
2) Else: depending on remaining space in the OMP and the size argument, set the type as ORAM or SCAN. Heuristics to select which are described below.

Post-allocation, users perform reads and writes to OblObjs through obl_read and obl_write, which instrument each operation based on the allocator's prescribed type, as shown in Figure 4. We describe the ORAM type below.

```
1 int obl_read(OblObj* o, int addr) {
2 #oblivious {
3   int ret; int tmp;
4   switch (o->type)
5   case OMP:
6     asm ("oaddi %0, %1, %2":
7     "=r" (tmp): "r" (addr), "r" (o->base));
8     asm ("ocld %0, 0(%1)":
9     "=r" (ret): "r" (tmp));
10    break;
11  case ORAM:
12    ret = oram("read", o, addr); break;
13  case SCAN:
14    for (int j = o->base, j < o->bound; j+=4) {
15      asm ("orld %0, 0(%1)":
16      "=r" (tmp): "r" (j));
17      asm ("ocmov %0, %1, %2)":
18      "+r" (ret): "r" (j==addr), "r" (tmp));
19    } break;
20  return ret; } }
```

Fig. 4: obl_read implementation (obl_write is analogous). `#oblivious` is short-hand to indicate that the body consists only of data oblivious instructions. `oram`'s implementation is discussed in Section IV-B6. "=r","+r" denotes output register; "r" denotes input.

obl_alloc decides on each allocation's type based on information returned by ocpuid. In the current design, ocpuid returns OSZ, the implementation-specific size of the OMP. Future implementations may also return richer information, such as machine cache sizes/etc. to make more informed decisions. Since size and branches/jumps in our OISA are Unsafe, the strategy selected for each allocation depends only on the program (which is Public) and the machine architecture. Lastly, we note that since the allocator makes decisions based on the order of previous allocations, more performance-sensitive

objects should be allocated first.

ORAM **and** SCAN **types.** When the oblivious object does not fit into the OMP, the allocator may implement it as an Oblivious RAM [49] (ORAM) or memory scan. ORAMs are randomized algorithms which implement oblivious memory in poly-logarithmic time. For ORAM, we use the ZeroTrace library [23] which is a data oblivious ORAM client written in our threat model. Depending on remaining OMP space, ZeroTrace's internal sub-structures (e.g., the ORAM stash and position map [23]) can be placed in the OMP, which we show can speedup the original ZeroTrace by $> 4\times$ (Section VII-C6). SCAN is a fallback that emulates oblivious memory using normal memory, and is implemented as a sequence of orld and ocmov instructions (Figure 4).

Pointed out by [1], when scan vs. ORAM is more efficient depends on the memory size and the allocator should take this into account based on the allocation size parameter.

C. Process-OS Interface

Processes interact with the OS through exception handling, context switching and system calls. We design the OISA to cause minimal friction with the existing OS-process interface.

1) Exceptions: Exceptions leak data-dependent conditions (e.g., when a divide by zero occurs) in programs [42], [1]. When an exception occurs on instructions with all Public operands, it is handled like a normal exception. When an exception occurs on an instruction with a Confidential operand, the hardware must mask that exception (e.g., by replacing the result with a canonical value and leaving the label unchanged). In this design, the adversary may learn an exception has occurred only if resulting data is explicitly declassified with ounseal.

2) Context switching: In the current design, the OMP (Section IV-B6) and register file labels are added as thread state. Labels in memory are mapped to pages in a region of virtual memory that cannot be accessed directly by the program (Section V-C1). While adding the OMP to thread state doesn't make context switching performance-prohibitive for the OMP sizes we consider in Section VII, it will for sufficiently large OMPs. We leave integrating the OMP into normal process virtual memory (e.g., by using the RISC-V VLS technique [69]), as future work. Finally, if the adversary is supervisor-level (Section II-B), we rely on the shielding system, e.g., SGX, to protect program data during context switches. For example, in an SGX setup [45], all data (Public and Confidential) would be stored within the SGX ELRANGE.

3) System calls: We rely on orthogonal software techniques to sanitize system call arguments [70], [43].

V. Implementation

This section describes how we prototyped our OISA on the RISC-V BOOM microarchitecture. Our design augments BOOM 'v2,' which is the most recent iteration of the BOOM design [30]. We give the exact parameters used for the architecture in Table II, which corresponds to the block diagram in Figure 5 and is a default BOOM configuration.

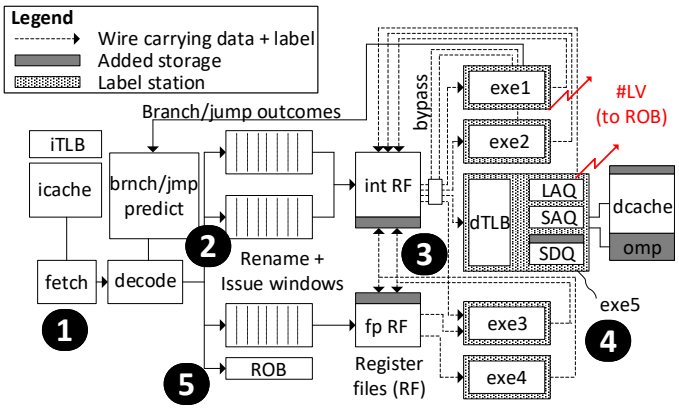

Fig. 5: RISC-V 'BOOM v2' pipeline [30]. 'exeXX' are execution units, and contain arithmetic/branch/etc units stated in Table II. Hardware modifications needed to support the OISA (Figure 2) are shown in the legend. No modifications are needed before the int/fp register files. Label stations are discussed in Section V-C2. 'omp' is the oblivious memory partition (Section V-B).

A. RISC-V BOOM Summary

We first summarize unmodified BOOM (referencing Figure 5). These details will be used for our implementation (this section) and formal analysis (Section VI).

First, multiple instructions are *fetched* each cycle ❶. Based on the current program counter (PC) and decoded instructions, multiple levels of branch/jump predictors issue predictions for fetched branches/jumps. Mispredicted branches/jumps are discovered in the execute stage, and cause subsequent speculatively decoded instructions to squash (Section II-A). Once *decoded*, instructions are added to the issue windows ❷ where they wait for their operands to be ready, at which point they are *scheduled* (possibly out-of-order) to execution units. Operands become ready when they are written (or written back) to one of two register files (RFs, for floats and integers) ❸, or when an execution unit finishes early and *bypasses* the result directly to the consumer instruction. RFs contain speculative and non-speculative data.

BOOM supports a configurable number of execution units ❹, each of which contains a configurable number of primitive arithmetic/branch/etc. units, shown in Table II. Each execution unit receives dedicated read/write ports to the RFs. Primitive arithmetic blocks may be *pipelined* (have input-independent latency) or *un-pipelined* (have input-dependent latency). Lastly, a load/store unit interfaces to the cache and decides whether load data should be read from the cache or store data queue (SDQ) which contains speculative stores (store-load forwarding). Loads may speculatively execute after stores whose address has not resolved [39]; address alias violations are caught and squashed at retire time. Finally, a reorder buffer (ROB) ❺ tracks in-flight instructions in-order to facilitate in-order commit (Section II-A).

The current BOOM does not currently support SMT/hyper-threading. We note that our OISA is compatible with an SMT-enabled machine and that the hardware mechanisms discussed below need not change to support SMT.

B. Support for New Instructions

Discussed in Section IV-B, most instructions in the OISA have exact counterparts in RISC-V, but with additional semantics/dynamic checks for Safe/Unsafe operands. These instructions reuse existing RISC-V encodings and have altered opcode/func fields to be identified during the decode stage. Several exceptions are oseal, unseal, orng, ocmov, ocld/ocst/ocpuid which don't have RISC-V counterparts (Figure 2).

We implement oseal and ounseal as the RISC-V addi instruction with the immediate field set to 0 (functionally a move operation), but with modified logic to set/clear label bits. As discussed in Section IV-B, ounseal must also serialize (execute non-speculatively) to prevent malicious declassification. Since BOOM already implements serializing instructions, we reuse that functionality for ounseal. Our prototype implements orng as a cryptographic PRNG (iterative AES core), although a hardware TRNG [71] may be used for a production design.

ocmov presents a challenge, as conditional move requires three operands (predicate, new value and old value) whereas no RISC-V integer instruction requires three input operands. To minimize ISA-level changes, we design a single ALU (in one execution unit) to serve ocmov instructions, and add a new RF port for that execution unit. We design this ALU to support bypassing. This design is low overhead and efficient. Having one execution unit support ocmov means we only need to add a single read port to the RF (not +1 per execution unit). Through bypassing, our design can execute back-to-back dependent ocmovs, one per cycle.

Finally, our current implementation implements the oblivious memory partition (OMP) for ocld/ocst as a quarantined region of the first-level data cache. We isolate a region of the cache using way partitioning techniques [72], which are a low-complexity mechanism to divide the cache into non-interferring regions as long as the region size is a multiple of the associativity (our first-level cache is 16-way; Table II). This design has low hardware overhead. If no process has allocated oblivious objects (Section IV-B6), OMP storage can be used as normal cache memory. While an ocld/ocst instruction is looking up the OMP, all concurrent cache lookups are stalled to avoid cache bank contention [55].

C. Tracking and Checking Labels

An important component in our OISA is checking and tracking Public/Confidential labels as data flows through the pipeline and signalling #LV when violations occur. Noted in Section IV-B, we track labels at word granularity.

1) Label storage: Labels must be stored alongside each word, where-ever each word resides in the processor. This includes the RF, the SDQ, the data cache hierarchy, and intermediate pipeline registers. In all of the above structures, we treat data label as an extra bit in each word. This makes it simpler to satisfy Rule IV.1: whenever a speculative or non-speculative instruction reads an operand, normal out-of-order processor dependency checking ensures the label is resolved.

Unfortunately, this strategy would require large changes to the DRAM/below memory levels because wider words would require wider DRAM lines and larger page tables. Thus, at the DRAM level, we store data and labels in separate disjoint pages and modify the hardware DRAM controller to join data and label into a widened cache line when on-chip (a similar scheme was used in [60]). This means any DRAM access in our system turns into two DRAM accesses.

2) Label checks: To satisfy Rules IV.2 and IV.3: once a consumer instruction indicates its intent to use an operand, that operand's label must be checked against the instruction opcode/-func fields, before the use occurs. We design a parameterizable hardware module called a *label station*, which wraps each BOOM execution unit, to administer these checks. The main observation enabling the label station design is that in BOOM, all operand-dependent processor state updates are signalled from the execution units. This makes it possible to implement a shim at the input of each execution unit to perform label checks, handle label violations/faults, and disable hardware optimizations on Confidential inputs.

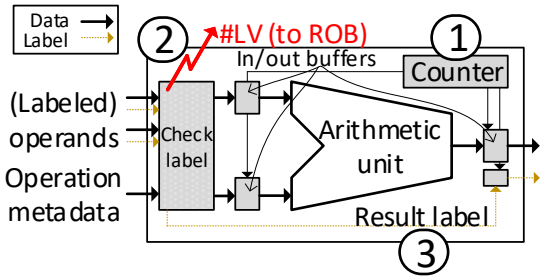

Fig. 6: Label station (Section V-C2) for an execution unit with one internal arithmetic unit. A real execution unit may contain multiple arithmetic units (Table II), in which case this logic is replicated as needed. Added hardware is shaded.

Specifically, the label station (visualized in Figure 6):

① (Rule IV.2: Confidential → Safe) Blocks access to/from arithmetic units so that any operation processing Safe operands takes the worst case time. This is implemented using input/output buffers (e.g., flip-flops), a timer (counter), and operand/label decode logic ("Check label" in the figure). Variable-time arithmetic units and their worst-case times are given in Table II. Lastly, any status bits set as a function of Confidential operands are set to canonical values.

② (Rule IV.3: Confidential → Unsafe) Checks each incoming operation for illegal label-operand violations, and signals #LV when violations are detected. All checks are performed before operands are forwarded to the execution unit. If any violation is detected, the execution unit does not receive the operation and an #LV signal is sent to the ROB, where it is interpreted as a violation (squash) or a fault (termination, #LF), respectively.

③ (Label propagation) Computes the result label based on operand labels and stages the label to travel with the result when it writes back to the RF or exits early via bypass.

Label stations are parameterized at design-time based on what functionality is actually needed. For example, Execution unit 2 (Table II) only supports Safe-operand arithmetic and therefore doesn't need logic to enforce Rule IV.3 (Confidential → Unsafe). Hence, this logic is pruned away at hardware synthesis time.

VI. Security Analysis

We will show that the OISA provides a basis for satisfying *Oblivious*[BitCycle] (Section III-A) by proving its security over an *abstract* out-of-order, speculative machine (AOOM), and arguing that this abstract machine can be reduced to real hardware such as the BOOM.

A. Takeaways and Main Insights

The takeaway from the analysis is that the OISA provides a basis to prove (computational) non-interference on an out-of-order processor with speculative execution. Importantly, we achieve this result *while* allowing hardware optimizations, such as branch predictors, to remain enabled and *without* requiring those structures to be partitioned or periodically flushed.

Informally, for this result to hold we need to show that (a) each instruction's visible execution and (b) the sequence of instructions that are executed is independent of Confidential data. (a) follows by definition, given Rules IV.2-IV.4, and is enforced by label stations in our implementation (Section V-C2). A key insight is that by performing these local checks, the hardware is also able to satisfy (b), and further that neither the hardware nor the analysis needs to treat transient and non-transient [11] instructions differently. This is non-trivial. Even given a correctly written program, prediction, speculation and subsequent squashes cause the dynamic sequence of executed instructions to depend on extra-program, even adversarially directed, effects (e.g., priming predictors [12]).

We give intuition on how to address (b) here; details follow in later subsections. To start, the OISA guarantees by design *that the inter-instruction program counter (PC) never becomes a function of ("tainted by") Confidential data.*[7] We likewise assume that predictor structures start off in a state that does not depend on Confidential data. Then, in the absence of an active adversary, we can prove that program behavior (with all hardware optimizations, such as speculative execution, enabled) is independent of Confidential data. Finally, we can model active adversaries without changing the analysis: since the program in isolation satisfies (a) and (b), the adversary's strategy in how to influence program behavior (e.g., by priming predictors) must also only depend on Public information, meaning that predictors/etc. remain untainted despite the adversary's strategy. Putting it together, we can show that none of the OISA, the adversary, or other Public events on the processor taint the PC, implying (b) as a corollary.

B. ISA Level

The following analysis assumes the OISA disables the ounseal instruction (Section IV-B) unless otherwise stated.

1) Abstract machine basics: The functional model for AOOM is given in Algorithm 2, with notations/helper functions explained in Table I and Algorithm 1. Our goal was to keep the model as simple as possible, while capturing core features. Specifically, the abstract machine: (1) has a 3-stage pipeline $\{\mathsf{Fetch}, \mathsf{Execute}, \mathsf{Retire}\}$ where each stage is atomic and takes one unit of time, (2) has four instruction types $\{\mathsf{Arithmetic}, \mathsf{Branch}, \mathsf{MemLoad}, \mathsf{MemStore}\}$, (3) has infinite fetch bandwidth and execution units, (4) can be parameterized as an in-order or out-of-order/speculative machine. Which instruction types support Safe/Unsafe operands are encoded as conditionals checking operands for label violations (#LV). We explain how to extend the model (e.g., to account for variable latency instructions, cache, limited execution units, more pipeline stages, etc.) in Section VI-C.

TABLE I: Notations and simple helper functions.

$\lvert T\rvert$	Returns number of elements in T
$T[i:j]$	Returns items with index i to j (inclusive)
λ	Public program
$\mathsf{Fetch}, \mathsf{Execute}, \mathsf{Retire}$	Instruction stages
$\mathsf{Arithmetic}, \mathsf{Branch}$ $\mathsf{MemLoad/Store}$	Instruction types
$\mathsf{stage}, \mathsf{pc}, \mathsf{squash}, \mathsf{update}$	Trace entry format
$\mathsf{Write}(\mathsf{addr}, \mathsf{data}, \mathsf{label})$	Token denoting write to program memory
$Proj(T)$	Trace with updates removed
$arg_i(\mathsf{pc}, \lambda), dest(\mathsf{pc}, \lambda)$	Returns instruction operand/dest fields
$op(\mathsf{pc}, \lambda)$	Returns instruction's implied arithmetic op
$T.append(e)$	Append e to end of of T
$type(\mathsf{pc}, \lambda)$	Return instruction at pc's type (Branch, etc)
$done(e, \lambda)$	Returns true if $e.\mathsf{stage} = \mathsf{Retire}$ and $e.\mathsf{pc}$ is the stop PC given λ
SCHEDULE, PREDICT	Instruction scheduler and predictor functions

2) Execution traces: The abstract machine AOOM takes as input a program λ, Public input x and Confidential input y and generates a trace T where each entry T_t tracks a stage of each instruction as it executes on the machine. That is, the t-th element in T is a 4-tuple:

$$T_t = (\mathsf{stage}_t, \mathsf{pc}_t, \mathsf{squash}_t, \mathsf{update}_t).$$

stage_t denotes the instruction's stage $\{\mathsf{Fetch}, \mathsf{Execute}, \mathsf{Retire}\}$. pc_t denotes the instruction address/program counter. Different stages for the same logical instruction share the same pc. If $\mathsf{stage}_t = \mathsf{Execute}$, $\mathsf{squash}_t = \{\mathsf{true}, \mathsf{false}\}$ denotes whether the instruction caused a squash during speculation (Section II-A) or due to a label violation #LV (Section IV-A2). If $\mathsf{stage}_t \neq \mathsf{Execute}$, $\mathsf{squash}_t = \mathsf{false}$. $\mathsf{update}_t = \mathsf{Write}(\mathsf{addr}, \mathsf{data}, \mathsf{label})$ where Write is a token denoting whether program memory was written, and with what addr, data and label. The Public label is logic 0, Confidential is logic 1. If no write occurs, $\mathsf{addr} = \perp$.

3) Modeling time: In our abstraction, entries in T are ordered in time as $\mathsf{time}(T_i) \leq \mathsf{time}(T_{i+j})$ for $i, j \geq 0$ where time is a metric for real time (e.g., clock cycles). That is, multiple events may occur in the same clock cycle (as in a real processor) or be separated far apart. Therefore, stage_t and $type(\mathsf{pc}_t, \lambda)$ allows us to model contention in different pipeline stages for different instruction types.

4) Modeling out-of-order and speculative execution: A key feature in our analysis is that AOOM is parameterized by two functions, SCHEDULE and PREDICT. SCHEDULE represents control logic in a real processor and decides which stage of which instruction should be evaluated next. It takes as input the program λ and $Proj(T)$, a projection of T that removes update from each entry, i.e.,

$$Proj(T) = \{e.\mathsf{stage}, e.\mathsf{pc}, e.\mathsf{squash} \textbf{ for } e \in T\}$$

Importantly, $Proj(T)$ constrains scheduling to not be a function of program data (i.e., e.update) beyond the sequence of present/past fetched instructions (e.stage, e.pc) and whether those instructions result in a squash (e.squash). SCHEDULE

[7] This follows from the semantics of Safe operands. Similar requirements on not tainting the PC govern prior work [65].

outputs an index idx $\in [0, |T|)$ or $\perp$. If idx $= \perp$, the machine will fetch the next instruction. If idx $\neq \perp$, the machine will evaluate the next stage for the instruction at T[idx]. PREDICT represents branch/jump predictor logic, takes the same inputs as SCHEDULE and outputs the predicted next PC. W.l.o.g. we assume SCHEDULE and PREDICT are deterministic.[8]

Importantly, SCHEDULE and PREDICT are representative of modern processors and allow us to model simple in-order processors to advanced out-of-order speculative processors (details on this claim related to BOOM are in Section VI-C). The only assumption we will make is that SCHEDULE respects in-order Fetch and Retire, as done by machines today.

5) Modeling machine state: The current machine state at some point idx in the trace is determined based on the trace prefix from 0 to idx. This includes program state (register file, cache, etc.) and intermediate pipeline/machine state. Program state is calculated based on *mem* (Algorithm 1). We merge the register file and other memory into a single memory for simplicity. Data always travels with its label, which models Rule IV.1. As mentioned in Section VI-B2, pipeline state (e.g., flip-flops/SRAM not included in program state) is modeled by the sequence of PCs and stages in the trace.

6) Proof of Security: We now prove that the abstract model AOOM satisfies Definition III.1 with respect to the following observability function WordStage.

Definition VI.1. *(*WordStage *observability: Public data and labels at* ***word****-spatial granularity,* ***instruction*** ***stage****-level temporal granularity) Given* $T = \mathsf{AOOM}(\lambda, x, y)$*,*

$$\mathsf{WordStage}(T) = \{e.\mathsf{stage}, e.\mathsf{pc}, e.\mathsf{squash}, h(e) \textbf{ for } \mathsf{e} \in T\}$$

where $h(e)$ *returns* e.update *(unmodified) if* e.update.label $=$ false*, and returns* Write$(e.\mathsf{addr}, \perp, \mathsf{true})$ *otherwise.*

That is, WordStage only removes write data from the trace if the label corresponding to that data is Confidential. Satisfying Definition III.1 with the WordStage function implies the strongest level of privacy with respect to our abstract machine, and implies that the machine's pipeline utilization, PC sequence, set of squash events, and state w.r.t. Public data is independent of Confidential data. We proceed to show Theorem 1:

Theorem 1. *Oblivious*[WordStage, AOOM] *holds.*

Proof: Let $T = \mathsf{AOOM}(\lambda, x, y)$ and $T' = \mathsf{AOOM}(\lambda, x, y')$ where x is an array of Public data and y, y' are arrays of Confidential data. We must show $\mathsf{WordStage}(T) \simeq \mathsf{WordStage}(T')$. We proceed using induction. Line numbers refer to Algorithm 2. The base case holds throughout *meminit* (Line 1, defined in Algorithm 1) since $|y| = |y'|$. Now assume $\mathsf{WordStage}(T[0:n]) \simeq \mathsf{WordStage}(T'[0:n])$. We know SCHEDULE (Line 3) will return the same idx for both executions because $Proj(T[0:n]) = Proj(T'[0:n])$ by the induction hypothesis and λ is fixed (Section VI-B4). We now proceed by cases depending on idx:

Case 1 (idx $= \perp$)**.** A new instruction will be fetched for both executions. We know PREDICT (Line 5) will return the same PC across both executions, following the same logic as SCHEDULE. By Line 6, it is clear the induction step holds.

Case 2 (T[idx].stage $=$ Fetch)**.** The instructions at idx in T and T' will be executed. There are 4 cases in Execute, depending on the instruction type $type(\mathsf{pc}, \lambda)$, and a careful inspection shows that each satisfies the induction step. Of note, the only possible deviation between T and T' is over update.data, in the case of Arithmetic and MemStore instructions when update.label $=$ true, which is allowed by Definition VI.1.

Case 3 (T[idx].stage $=$ Execute)**.** The instructions at idx in T and T' will be retired. By Line 13, it is clear the induction step holds.

■

7) Extensions to randomized cryptographic algorithms: It is straightforward to extend the above analysis to support randomized cryptographic algorithms such as ORAM [49], [23]. For example, ORAM client logic can be written data obliviously to satisfy *Oblivious*[WordStage, AOOM] [23], [73]. What is left is to show how the visible ORAM access pattern—which forms a subset of the trace—satisfies computational indistinguishability [49]. This reduces to the security of the ORAM protocol itself and to the OISA's mechanism to declassify private data, i.e., ounseal. For the latter, since ounseal is a serializing instruction, we know private randomness will be exposed if and only if it is intended by the protocol.

Algorithm 1: Helper functions *meminit* and *mem*.

```
/* fill memory w/ Public x, Confidential y      */
function: meminit(x, y)
1  T := [];
2  for x_i ∈ x do
3      T.append((Execute, ⊥, false, Write(i, x_i, false)))
4  for y_i ∈ y do
5      T.append((Execute, ⊥, false, Write(|x| + i, y_i, true)))
6  return T;
7
/* return coherent memory snapshot, given T.
   Note, an instruction that is squashed by
   another instruction may still create visible
   state changes in the window of time before
   the other instruction reaches Execute.       */
function: mem(T)
8  T' = T with all squashed instructions (trace entries) removed.
   That is, remove from T any entry that occurs in between the
   Fetch and Execute stage of an instruction I if I satisfies
   I.stage = Execute ∧ I.squash (inclusive);
9  mem := [⊥ for t ∈ T'];     // |T'| upper-bounds mem size
10 for x_i ∈ T' do
11     up := x_i.update;
12     if up.addr ≠ ⊥ then
13         mem[up.addr] = up.data, up.label;
14 return mem;
```

C. Implementation Level

We now map our ISA-level security analysis (Sections IV-B and VI-B) to our prototype on BOOM (Section V), referred to as BOOM.

[8] Heuristics based on randomness can be modeled with an additional seed input.

Algorithm 2: Abstract machine definition. As in Figure 2, the Public label is logic 0, Confidential is logic 1.

```
   function: AOOM(λ, x, y)
1  T := meminit(x, y);                                  // initialize memory
2  while !done(T[|T| − 1], λ) do
3      idx := SCHEDULE(Proj(T), λ);
4      if idx = ⊥ then                                  // Fetch new instr
5          pc := PREDICT(Proj(T), λ);
6          T.append((Fetch, pc, false, Write(⊥, ⊥, false)));
7      else
8          pc := T[idx].pc;
9          stage := T[idx].stage;
10         if stage = Fetch then                        // Execute instr
11             T.append(execute(Execute, pc, T, λ));
12         else if stage = Execute then                 // Retire instr
13             T.append((Retire, pc, false, Write(⊥, ⊥, false)));
14 return T;
15
   function: execute(stage, pc, T, λ)
16 update := Write(⊥, ⊥, false); squash := false;
17 arg0,data, arg0,label := mem(T)[arg0(pc, λ)];
18 arg1,data, arg1,label := mem(T)[arg1(pc, λ)];
19 if type(pc, λ) = Arithmetic then
20     data := arg0,data op(pc, λ) arg1,data;
21     label := arg0,label ∨ arg1,label;
22     update := Write(dest(pc, λ), data, label);
23 else if type(pc, λ) = Branch then
24     if arg0,label ∨ arg1,label then
25         squash := true;   // #LV: Confidential->Unsafe
26     else
27         fidx := index of Fetch for current instr in T;
28         guess := direction for PREDICT(Proj(T[0 : fidx]), λ);
29         actual := arg0,data op(pc, λ) arg1,data;
30         squash := guess ≠ actual;                    // mispredict
31 else
32     if arg0,label then
33         squash := true;   // #LV: Confidential->Unsafe
34     else
35         if type(pc, λ) = MemLoad then
36             data, label := mem(T)[arg0,data];
37             addr := dest(pc, λ)
38         else if type(pc, λ) = MemStore then
39             data, label := arg1,data, arg1,label;
40             addr := arg0,data
41         update := Write(addr, data, label)
42 return stage, pc, squash, update;
```

1) Threat vectors in unmodified BOOM: Unmodified BOOM hardware (Section V-A) supports speculation over branches, jumps and unresolved store instructions (Vectors 1-3; c.f. Section III-B) as well as arithmetic units with input-dependent timing (Vector 6, Table II).[9] Our implementation of the OMP (Section V-B) is also susceptible to cache bank contention (Vector 5) because it uses space in the data cache.

[9]We note BOOM also supports load/store forwarding but is not susceptible to Vector 4 because the data TLB is accessed sequentially before checking the SAQ (Section V-A).

2) Securing BOOM: Recall, the primary hardware mechanisms we added to get security are dynamic information flow tracking (Section V-C1), label stations per execution unit to implement Safe/Unsafe operand semantics (Section V-C2), and logic to isolate the OMP (Section V-B).

In Section VI-B, we proved *Oblivious*[WordStage, AOOM]. We show how to use the proof to argue *Oblivious*[BitCycle, BOOM]—i.e., cycle-level security of our implementation—which implies that Vectors 1-3 and 5-6 are blocked. There are two steps: (1) mapping AOOM to BOOM and (2) mapping WordStage to BitCycle. We argue these reductions should go through, below, but with the caveat that our argument is best effort since we have only performed a careful by-hand inspection of the BOOM design. Using formal/automated methods to improve design confidence is important future work.

Step 1: mapping AOOM to BOOM. We kept AOOM simple to illustrate key points for presentation, but can make the model more sophisticated to better represent our actual implementation. In particular: relative to BOOM, AOOM has (a) an idealized SCHEDULE/PREDICT, (b) an unlimited number of execution units without dependencies, (c) less instruction types, (d) less pipeline stages, (e) fixed latency ("atomic") arithmetic/memory operations, (e) no support for jump/memory speculation, and (f) no support for the OMP.

For (a): we note that BOOM's actual scheduler/predictors operates over a subset of the inputs given to SCHEDULE and PREDICT. For example, BOOM's branch/jump predictors take as input the sequence of fetched PCs (speculative and non-speculative) and branch outcomes for retired instructions [30]. The sequence of PCs is contained in $Proj(T)$ and branch outcomes can be extrapolated from $Proj(T)$ given the program λ (Section VI-B4). The issue/scheduler logic is similar: employing data-independent information such as instruction age in the window and public instruction dependencies.

For (b): limited execution units (structural hazards) and instruction dependencies (data hazards) can be modeled as additional rules in the SCHEDULE function. For example, SCHEDULE may not consider executing an in-flight instruction until its dependencies given in-flight instructions (recent trace entries) have executed. Note that when an instruction is executed with Confidential data, the operand consuming that data is either Safe (in which case execution behavior is independent of the data) or results in #LV (which causes an exception independent of the data).

For (c) and (d): it is straightforward to add extra stages (e.g., BOOM's issue stage) and instruction types. We note that the 4 types (Arithmetic, etc.) in the analysis cover the major cases in OISA (Section IV-B).

For (e): variable arithmetic can be modeled by creating additional execute stages Execute^i for $i < L$. To match our label station hardware (Section V-C2), the limit L can be set to a constant based on the operand label and the instruction type. Caches can similarly be modeled by allowing L to become a function of the sequence of previous memory addresses. Additional types of speculation (e.g., jump and memory speculation) can be modeled using Branch type instructions.

For (f): it is easy to model the OMP (Section IV-B6) in a fashion analogous to MemLoad and MemStore, as hardware

mechanisms moderating access to the OMP ensure that the OMP satisfies non-interference w.r.t. other cache lookups.

Note, we also don't model RF or cache compression (Vectors 8-9) as these optimizations don't appear in BOOM. They can similarly be modeled with addition #LV checks on data at rest.

Step 2: mapping WordStage **to** BitCycle**.** It is straightforward to extend Theorem 1 to include time (Section VI-B3), meaning the analysis can satisfy a "WordCycle" observability function as opposed to WordStage. We have performed a careful audit of our BOOM hardware prototype (analyzing logic paths through major flows in the processor at cycle level) to ensure intermediate storage elements (e.g., pipeline flip-flops) correspondingly carry Confidential data in data-independent cycles. We have not found violations to this rule outside of one exception (given below).

The main insight that enables the correspondence is that stages (e.g., Fetch, Execute) in our abstract model (Section VI-B2) correspond to intermediate storage elements. That is, each instruction stage corresponds to storage elements, e.g. pipeline registers in functional units, and the proof shows how the sequence of stages in the trace is invariant to Confidential input. This means that modeling AOOM with additional stages/sub-stages (e.g., rename, issue, Execute^i, etc.) provides a basis for modeling complex machines at the flip-flop and cycle level. We leave automated analyses on the design (e.g., using [63]) as future work.

The one exception (noted above) is label stations. We assume label stations are implemented correctly, meaning a software adversary cannot view flip flop-level label state within a label station, while that unit is processing Confidential data, or within the OMP. This is a minor detail, handled in the analysis by removing state bits in BitCycle when units are processing Confidential data. A similar principle is applied in showing security for Execution Leases [27].

Lastly, we note that WordCycle is equivalent to BitCycle in modern processors, including the BOOM. In modern ISAs like RISC-V, operations occur at word granularity meaning that labels only need to be tracked at word granularity.

VII. Evaluation

We now evaluate the OISA in terms of area overhead (given our prototype on RISC-V BOOM) and performance over data oblivious workloads. We also show two case studies, showing how the OISA secures and accelerates constant time cryptographic code and memory oblivious libraries.

A. Methodology

We evaluate our system through hardware prototyping to show area overheads and software simulation to show performance.

1) Hardware prototyping: We build on top of the open-source BOOM design [30] which is written in the Chisel hardware description language [74]. We parameterized the prototype according to Table II and synthesized the design using a 32 nm commercial process and the Synopsys flow. We report standard cell (logic cell) area for logic and flip-flops post-synthesis, and report SRAM area using the widely used Cacti tool [75]. BOOM maps the instruction/data caches/TLBs and branch predictor tables to SRAM. Remaining storage structures (e.g., the SDQ, RFs) are mapped to flip-flops. The BOOM word width is 64 bits.

TABLE II: RISC-V BOOM parameters we use for our prototype and evaluation. Arithmetic units with a '(xx)' next to their name are un-pipelined (variable latency), where 'xx' denotes the worst-case latency. The prefix 'i' denotes integer, 'f' denotes floating point. **CondMove** and **Omp** denote logic for ocmov and the oblivious memory partition (Section IV-B), respectively, and are only present on our modified BOOM.

Core μarch	out-of-order, speculative
Fetch/issue width	4 instructions fetched/issued per cycle
Execution unit 1	iALU, Branch, iMul, iDiv (6-66)
Execution unit 2	iALU, **CondMove**
Execution unit 3	IntToFP casting
Execution unit 4	fAdd, fMul, fDiv (5-21), fSqrt (5-29), FPToInt casting
Execution unit 5	Load/store + **Omp** (memory unit)
L1 I/D cache	32 KB, 4 way/64 KB, 16 way; 64 B cache lines
I/D TLB	16/32 entries

2) Software simulation: The BOOM hardware only features a single-level cache, whereas commercial machines feature two- or three-level caches to reduce traffic to DRAM. Thus, to measure more realistic performance figures for our system we use Multi2Sim [76], parameterized to match Table II as closely as possible. For all experiments, we use a 256 KB 4-way level 2 cache (that is shared by data and instructions) and a 2 MB 16-way level 3 cache. This configuration is similar to a single slice on an Intel Skylake machine.

3) OMP usage: We use a 32 KB OMP (Section IV-B6) that is built into the level 1 data cache. This is sufficient to store ORAM sub-structures (Section IV-B6) and also big enough to fit tables for constant time cryptographic routines (e.g., AES T-tables and RSA multiplier tables). Some workloads do not benefit from the OMP (e.g., some do not have data-dependent memory access patterns). In this case, a bit in thread state disables the OMP to recover cache space.

B. Hardware Prototyping and Area Results

We show area results for unmodified BOOM and BOOM extended to support our OISA in Table III. Our prototype supports all instructions in Section IV-B and Figure 2. The main hardware components needed to support the OISA are storage for DIFT, logic/storage for label stations, logic to partition the OMP, and a random number generator for orng (Section V). For structures that need to store labels, we store those labels alongside the data in whatever medium the data was stored in. That is, labels in data cache are stored in SRAM, labels in the SDQ and register files are stored in flip-flops. The largest single area overhead comes from an iterative AES core that we downloaded from OpenCores [77] to implement orng. This unit has area 10,935 um^2 (3% of the logic area for the unmodified BOOM), and can be replaced by a hardware TRNG (whose area is negligibly small [71]) in a production design.

The takeaway is that hardware overheads are tolerable, both on the logic and SRAM side, showing the practicality of the proposal on advanced commercial-class machines.

TABLE III: Area (um^2) for baseline and modified BOOM cores.

	BOOM	BOOM + OISA	Overhead
Logic	363,900	388,658	6.80%
SRAM	384,232	391,291	1.84%
Total	748,132	779,949	4.25%

TABLE IV: Benchmarks and input data sizes for comparing insecure, oisa and oisa_omp.

Name	Implementation	Data size (small / large)
mat. mult	data oblivious by default	256x256 / 1024x1024
neural network	""	64-1K-8 / 1024-32K-256 (2 layers)
findmax	""	8K / 1M integers
sort	bitonic-sort (oisa), data obl. merge-sort (oisa_omp)	4K / 256K integers
pagerank	GraphSC [28]	1K / 16K nodes
binary search	memory scan (oisa), obl. memory (oisa_omp)	8K / 16M integers
kmeans	obl. memory for histogram	64/256 clusters, 4K/32K points
heap push	ODS [68]	8K / 32M integers in heap
heap pop	ODS [68]	8K / 32M integers in heap
sparse dijkstra	ObliVM [66]	256 / 4K vertices

C. Performance Results

We now perform studies to evaluate the performance overhead of running data oblivious code securely, with and without the oblivious memory partition.

1) Comparison systems: We compare two systems—oisa and oisa_omp—to a baseline insecure system. All three systems use the same microarchitecture (Table II). Benchmarks run on insecure are written in a non-data oblivious fashion (i.e., without the constraints in Section III-B). Benchmarks run on oisa are data oblivious, and written using only instructions in Figure 2 except ocld/ocst (the oblivious memory extension; c.f. Section IV-B6). Thus, oisa will be similar performance-wise to existing data oblivious codes, e.g., Raccoon [1], which don't have access to an OMP. Benchmarks run on oisa_omp use all instructions in Figure 2 including ocld/ocst.

2) Workloads: We evaluate a suite of common workloads (Table IV) which have previously been written and evaluated data obliviously [1], [68], [28], [66] on existing x86 machines. These codes are divided into three categories. First, codes that are nearly data oblivious in their default form (mat mult, neural network, findmax). Second, codes that rely heavily on data oblivious sort as a subroutine (sort and pagerank). Third, codes that rely heavily on oblivious memory (binary search, kmeans, heap, dijkstra). We will also perform case studies showing our proposal's applicability in two additional important settings—constant time cryptography and oblivious memory—in Sections VII-C5 and VII-C6.

3) Data set sizes: For each benchmark, we evaluate 'small' and 'large' data sizes. 'small' indicates the largest input size that wholly fits into the 32 KB OMP (Section VII-A). We use this configuration for two reasons. First, to show the benefit of having an OMP. Second, to performance compare against prior work (Raccoon [1], which uses similar data sizes). Finally, we show the 'large' data size to illustrate overheads where program data does not completely fit into the OMP. In that case, we fallback to ORAM or SCAN as described in Section IV-B6.

4) Results: Figure 7 shows the overhead of {oisa, oisa_omp} × {small, large} relative to insecure. The main

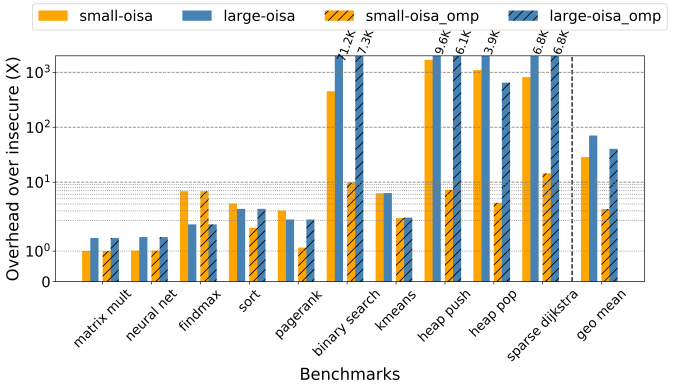

Fig. 7: Performance comparison between oisa and oisa_omp relative to insecure for small/large data sets.

takeaway is that oisa_omp achieves significant (8.8×/1.7× for small/large data sizes) speedup over oisa. Furthermore, oisa_omp has only 3.2×/40.4× slowdown relative to insecure on the same data sizes. This shows that our OISA makes data oblivious computing practical in cases where data fits in the OMP.

There are two avenues for future work. First, enhance the OMP to support larger sizes (e.g., beyond the level 1 data cache, see Section IV-C2). As we see on the large data set size, overhead for both oisa and oisa_omp can be large for workloads that depend on oblivious memory, as large data sizes cannot fit into the OMP. Second, engineer more sophisticated instructions supporting Safe operands. For example, sort is an important kernel in multiple data oblivious codes [78], [28], [19], [66]. An OISA can support an osort instruction with Safe operands directly, and use techniques such as hardware partitioning to speedup that operation.

5) Case study: constant time AES: An important commercial use-case for data oblivious code today is "constant time" cryptography. Many papers have demonstrated how unprotected codes—e.g., T-table AES [37] and naive modular exponentiation for RSA—leak privacy over microarchitectural side channels [4], [55], [54]. As a result, practitioners use slower codes to improve security—e.g., S-box or bitslice AES [38] and montgomery ladder exponentiation for RSA.[10]

Our OISA provides a basis for running high-performance cryptography securely. To demonstrate the benefit, we compare the performance of T-table AES [37] (high performance, low security) vs. bitslice AES [38] (low performance, high security). For this study, we retrofit T-table AES using our ISA and store the T-tables in the OMP to prevent cache attacks (the rest of the code is naturally data oblivious). This gives us a high performance, high security code. The OISA can securely run both the fully unrolled code or a variant with a loop over the number of rounds, regardless of branch prediction accuracy (Section III-B). We argue that on commodity machines today, highly sensitive applications will have to resort to codes like bitslice AES.

Both codes are compiled with gcc using `-O3` optimizations. Relative to an insecure T-table AES code (insecure), our data

[10]Discussed in Section III-B, even hardened codes may be insecure due to subtle hardware optimizations.

oblivious T-table AES (oisa_omp) has a 2.17× slowdown, while bitslice AES has a 9.6× slowdown against the same baseline. Our slowdown relative to insecure is caused by the compiler not optimizing code around ocld instructions. Thus, oisa_omp can achieve even lower slowdown with better compiler support.

6) Case Study: ZeroTrace [23]: Beyond encryption, there is a rich literature to accelerate data structure operations data obliviously [68], [66], [21]. These schemes typically use oblivious memory as a subroutine. We now demonstrate how the OISA can speedup this subroutine by comparing our oblivious memory API to the original ZeroTrace [23] proposal. Discussed in Section IV-B6, our library combines ZeroTrace with the OMP to achieve speedup for different oblivious memory sizes.

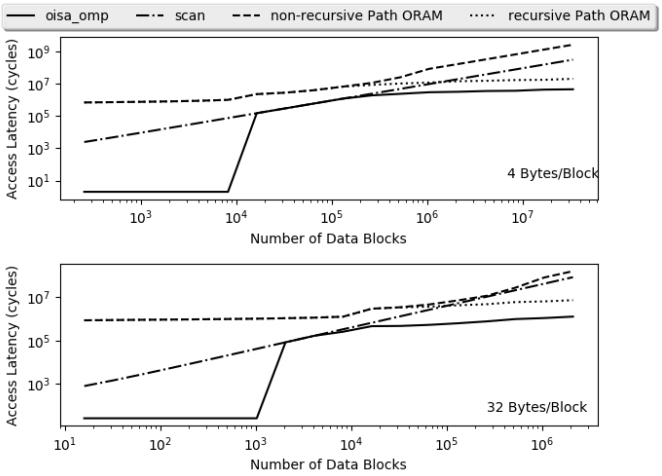

Fig. 8: Comparison between oblivious memory primitives. Scan is the SCAN code from Figure 4, shown for completeness. non-recursive/recursive Path ORAM are baseline ZeroTrace [23].

Results are shown in Figure 8. oisa_omp provides significant speedup in all size regimes. For small data, oisa_omp places the entire memory in the OMP, providing $O(1)$ ($>1000\times$ speedup) time access to that data. For larger data, oisa_omp uses SCAN or ORAM, depending on which strategy yields best performance, and places the ORAM stash in the OMP in the latter case. An important finding in the ZeroTrace paper is that stash management, written data obliviously, creates a performance bottleneck.[11] Since the stash does not grow as a function of the ORAM capacity, we can use the OMP to store the stash and manage it more efficiently, which allows us to improve over baseline ZeroTrace by $\geq 4.6\times$ in all regimes.

VIII. Related Work

Data oblivious stack. Beyond data oblivious code written for today's ISAs, there is a rich literature to improve algorithm/data structure [49], [29], [78], [28], [79], [80], [66], [81], [82], [68] performance in the *software circuit* abstraction. Additionally, there is rich literature to write (e.g., [67], [83]) and compile (e.g., [66], [84], [81]) programs to software circuits. An important observation is that, although many of these works target cryptographic backends such as garbled circuits, their underlying programming abstraction (software circuits) is very similar to the data oblivious abstraction. For example, bitwise crypto can be easily mapped to integer-wide operations. Thus, our proposal can be used as a secure hardware backend for these works.

Secure co-processors. Secure co-processor proposals Ghostrider [32] and Ascend [31] have the same security goal (Definition III.1) as this paper, but assume a course-grain observability function that only captures the processor's external pin activity (whereas this paper considers fine-grain observability; c.f. Section III-A). These proposals also assume simple processor pipelines and scheduling (e.g., one process per chip at a time). Relative to these works, our goal is to show how to retrofit *existing* high-performance machines to concurrently run sensitive and non-sensitive programs, which matches how programmers are writing data oblivious code today.

Architecture to mitigate side channels. There is a significant body of work aimed at blocking specific side channels inside (e.g., [85], [86]) and outside (e.g., [87], [88]) of speculative execution. We view these works as complementary in that they are implementations which can be used to realize various instructions with Safe operands. We note that these works typically do not satisfy Definition III.1 at cycle granularity as is, nor do they provide a composable method to block all (as opposed to specific) side channels without an additional layer such as an OISA.

ISAs for security, type systems for information flow guarantees. ISAs for security are not new; further Safe/Unsafe operands and the use of DIFT can be viewed as performing runtime checks between a simple type system and security lattice (e.g., see [89], [65], [63], [27]). Relative to these lines of work, we view our conceptual contribution as introducing new ISA abstractions and design principles that allow software/hardware designers to trade-off efficiency with implementation complexity, while leaving programmers with simple, portable security guarantees. We note that our choice of lattice and types was done for simplicity; an OISA may be combined with a more sophisticated lattice and set of operand functionalities.

Finally, we view GLIFT [65], [27] as a spiritual predecessor to this work. One of GLIFT's major insights is that at the logic gate level, implicit and explicit flows look very similar. We observe that the same is true in the data oblivious abstraction at the program level. This allows insight from GLIFT to carry over to our domain (e.g., GLIFT's bit-level checks/transition functions perform a similar purpose for bits as label stations perform for words; c.f. Section V-C2).

IX. Conclusion

This paper proposes an Oblivious ISA extension to enable secure and high-performance data oblivious computing in the data oblivious abstraction. We propose ISA principles, a concrete ISA, a complete prototype on an advanced microprocessor, and accompanying formal analyses for all of the above. Long term, we hope this paper serves as a step for writing and

[11] We note that an alternate ORAM, Circuit ORAM [73], was designed to avoid stash management overheads. Unfortunately, Circuit ORAM has worse bandwidth—$12*\log n$ vs. $8*\log n$ and $3.5*\log n$ for data size n—than Path ORAM, which relies on a stash. Since our oblivious memory extensions make stash management essentially free, our scheme based on Path ORAM will outperform Circuit ORAM.

running safe, portable and performant data oblivious code for sensitive applications.

Acknowledgements. We thank the anonymous reviewers for their feedback, and Mohit Tiwari for many interesting discussions. This work was funded through NSF awards #1725734 and #1816226, and an Intel ISRA.

REFERENCES

[1] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Security'15*.

[2] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*.

[3] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," *CoRR'17*.

[4] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *CT-RSA'06*.

[5] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," in *Security'14*.

[6] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*.

[7] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS'18*.

[8] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P'15*.

[9] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," *IACR'09*.

[10] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit." IACR'18.

[11] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses." CoRR'18.

[12] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *SP'19*.

[13] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *PKC'06*.

[14] D. J. Bernstein, "The poly1305-aes message-authentication code," in *FSE'05*.

[15] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," *IACR'05*.

[16] D. B. S. G. Ben A. Fisch, Dhinakaran Vinayagamurthy, "Iron: Functional encryption using intel sgx," in *CCS'17*.

[17] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *Security'16*.

[18] Z. L. L. K. Fahad Shaon, Murat Kantarcioglu, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *CCS'17*.

[19] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *NSDI'17*.

[20] S. Eskandarian and M. Zaharia, "An oblivious general-purpose SQL database for the cloud," *CoRR'17*.

[21] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P'18*.

[22] S. Tople and P. Saxena, "On the trade-offs in oblivious execution techniques," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (M. Polychronakis and M. Meier, eds.), Springer'17.

[23] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel sgx," in *NDSS'18*.

[24] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx," in *NDSS'18*.

[25] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P'09*.

[26] "Speculative execution side channel mitigations." https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf. Revision 1.0, January 2018.

[27] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *MICRO'09*.

[28] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *S&P'15*.

[29] "Bitonic sort." https://en.wikipedia.org/wiki/Bitonic_sorter.

[30] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovi, "Boom v2: an open-source out-of-order risc-v core," tech. rep., EECS Department, University of California, Berkeley, 2017.

[31] C. Fletcher, M. Van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *STC'12*.

[32] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *SIGPLAN Not.*, vol. 50, pp. 87–101, Mar. 2015.

[33] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shik, and V. Goyal, "Hop: Hardware makes obfuscation practical," in *NDSS'17*.

[34] J. Mclean, "Security models," in *Encyclopedia of Software Engineering*, Wiley & Sons, 1994.

[35] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Security'18*.

[36] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 557–574, USENIX Association, 2017.

[37] "T-table AES (OpenSSL)." https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c.

[38] "Bitslice AES (Bitcoin)." https://github.com/bitcoin-core/ctaes.

[39] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *ISPASS'04*.

[40] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *SIGPLAN Not.*, vol. 31, pp. 138–147, Sept. 1996.

[41] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," *CoRR'17*.

[42] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," February 2017.

[43] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *OSDI'16*.

[44] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *S&P'11*.

[45] Intel, "Intel Software Guard Extensions Programming Reference." software.intel.com/sites/default/files/329298-001.pdf, 2013.

[46] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *CCS'17*.

[47] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*.

[48] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "Eddie: Em-based detection of deviations in program execution," in *ISCA'17*.

[49] 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," *CCS'13*.

[50] M. Backes and B. Pfitzmann, "Computational probabilistic noninterference," *International Journal of Information Security*, 2004.

[51] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, Jan 2003.

[52] S. Gueron, "Efficient software implementations of modular exponentiation," *IACR'11*.

[53] Intel, "Intel Software Guard Extensions Software Development Kit." https://software.intel.com/en-us/sgx-sdk.

[54] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *CoRR'17*.

[55] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa," *IACR'16*.

[56] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *MICRO'98*.

[57] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," *SIGARCH Comput. Archit. News*, vol. 32, pp. 212–, Mar. 2004.

[58] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.

[59] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *MICRO'00*.

[60] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *SIGARCH Comput. Archit. News*, vol. 35, pp. 482–493, June 2007.

[61] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, pp. 85–96, Oct. 2004.

[62] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, pp. 236–243, May 1976.

[63] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *SIGPLAN Not.*, vol. 50, pp. 503–516, Mar. 2015.

[64] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[65] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGARCH Comput. Archit. News*, vol. 37, pp. 109–120, Mar. 2009.

[66] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *S&P'15*.

[67] D. Darais, C. Liu, I. Sweet, and M. Hicks, "A language for probabilistically oblivious computation," *CoRR'17*.

[68] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," *IACR'14*.

[69] H. Cook, K. Asanovi, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," tech. rep., 2009.

[70] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *S&P'09*.

[71] V. Fischer, "Random number generators for cryptography (design and evaluation." https://summerschool-croatia.cs.ru.nl/2014/slides/Random%20Number%20Generators%20for%20Cryptography.pdf.

[72] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *TACO'12*.

[73] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," *IACR'14*.

[74] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *DAC'12*.

[75] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[76] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *PACT'12*.

[77] "Open cores." https://opencores.org/.

[78] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *ASIA CCS'13*.

[79] J. Doerner, D. Evans, and abhi shelat, "Secure stable matching at scale," *IACR'16*.

[80] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," *IACR'17*.

[81] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *S&P'15*.

[82] S. Zahur and D. Evans, "Circuit structures for improving efficiency of security and privacy tools," in *S&P'13*.

[83] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," *SecDev'17*.

[84] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," *IACR'15*.

[85] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO'18*.

[86] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *MICRO'18*.

[87] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA'16*.

[88] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Security'17*.

[89] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *CCS '18*.