Creating Foundations for Secure Microarchitectures with Data-Oblivious ISA Extensions

Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, Christopher W. Fletcher
University of Illinois at Urbana-Champaign

1 Introduction

A, arguably *The*, central problem in Secure Computer Architecture today is how to reason about security amidst the sea of different microarchitectural side channel attacks. The prevailing approach to stop these attacks is to block leakage stemming from one hardware structure at a time. For example, by partitioning or randomizing the cache layout, we block (or at least aggravate) cache timing attacks. Yet, many hardware structures have been shown to leak secrets—from the cache to the branch predictors [5], speculative execution [8], port contention [1], arithmetic unit timing [2] and more. Given the many avenues to leak a secret, it is paramount to explore *holistic defenses* that provide a basis to block leakage through *all* hardware structures.

In this direction, our paper proposes ISA design principles for what we call *Data-Oblivious ISAs* (OISAs). The key idea with an OISA is to explicitly but abstractly specify security policy at the ISA level and, importantly, to decouple this policy from a processor's implementation. Analogous to a traditional ISA, this enables an OISA to serve as a portable security-centric abstraction for software, while enabling security-aware implementation and optimization flexibility for hardware.

The OISA proposed in the paper annotates what data is *Confidential* and what instruction operands are *Safe*. Inspired by information flow policies (in particular the classic policy $High \nrightarrow Low$), the hardware dynamically enforces *that Confidential data is never passed to Unsafe operands*, i.e., $Confidential\ data \nrightarrow Unsafe\ operands$. Informally, "Safe" in the paper means "does not create a microarchitectural side channel as a function of the operand" (we also provide formal definitions), but other notions of safety can be retrofitted into the implementation without changing the OISA or the programs that sit on top of it.

OISAs enable high security, portability and efficiency. Consider a simple example OISA instruction: a load with a *Safe* address operand. Security- and portability-wise, the OISA guarantees that when the load executes, the address will not leak through microarchitectural side channels. Depending on the microarchitecture, this may require closing side channels through the cache, TLB, etc. That is, security isn't tied to closing a specific side channel and the programmer works with a simple, portable guarantee across microarchitectures. On the efficiency side, each microarchitecture can choose how to implement the Safe load operation in whatever way maximizes performance while preserving security (e.g., by micro-coding the load into simpler Safe operations [2], or using hardware partitioning [11] or using cryptographic techniques [9]).

Safe loads are just one example. More generally, deciding which instruction operands to designate as Safe opens a new, rich ISA design space which trades-off performance and hardware complexity.

Beyond formulating design principles for OISAs, the paper proposes a concrete OISA extension built on top of RISC-V, implements (and open-sources) that OISA extension on the BOOM out-of-order (OoO) speculative RISC-V core [3], and provides a formal analysis showing how the OISA provides a basis to achieve non-interference ("zero privacy leakage") on an abstract out-of-order speculative machine. Crucially, the security analysis and principles are robust to modern attacks. Case in point, the paper's formal analysis shows how the OISA soundly defeats speculative execution attacks (such as Spectre [8]) without introducing special case reasoning.

To our knowledge, this is the first proposal to that provides a basis to block all traditional side channel *and* speculative execution attacks on commercial-class microarchitectures.

2 Motivation: Secure and Efficient Data-Oblivious Programming

The OISA project came about by asking the following question: *is it possible today to write microarchitectural side channel-free programs on modern microarchitectures?*

The answer is no. Consider the most conservative approach used by practitioners, called *data-oblivious programming*.[1] In a nutshell, a data-oblivious program is one whose hardware resource usage is independent of the program's inputs. To write such programs, the guidelines are to use only simple instructions, or otherwise ensure that complex instructions do not receive Confidential data as operands. For example, simple bitwise math is allowed, but memory operations/branches with Confidential data as addresses/predicates are not (out of fear of, e.g., cache-based/control flow-related side channels).

Despite being extremely conservative, the above guidelines fail in light of ISA-invisible microarchitecture-specific optimizations. For example, on one microarchitecture a simple integer addition might be safe (e.g., implemented as a single-cycle operation whose timing is independent of its inputs) while on another it might be unsafe (e.g., implemented as a bit-serial operation that skips runs of 0s to save time). The paper describes 11 like optimizations which have been proposed in the literature, or are otherwise known to be implemented already, which break data-oblivious program security. These include data-in-use optimizations (such as data-dependent arithmetic) and data-at-rest optimizations (such as cache compression).

In particular, the paper points out for the first time that speculative execution breaks data-oblivious program security, by steering execution so that Confidential data is consumed by an instruction whose execution can leak privacy. This is non-trivial to see for realistic programs, given the conservative guidelines used to write data-oblivious code. For example, consider data-

[1]Data-oblivious programming goes by several other names, e.g., "constant-time programming" and "programming in the circuit abstraction," depending on the community.

oblivious decryption:

```
for (i = 0; i < NUM_ROUNDS; i++)
  state = OblDecryptRound(state, rkey[i])
leak(state)
```

That is, perform a fixed number of decryption rounds, where each round works on a part of the secret key (`rkey`) and incrementally updates the round state (`state`). Here, we assume that `OblDecryptRound`, the round logic, is data oblivious. `leak()` is a proxy for an instruction that reveals its argument over a microarchitectural side channel.

This program is legal data-oblivious code: the branch outcome in each iteration is public information, the round logic is data oblivious, and only the plaintext is meant to be revealed after decryption is complete. Yet, unwanted privacy leaks because benign mispredictions can cause the round logic to exit early. In this example, an early mispredict of "not taken" allows the attacker to see `state` before all rounds complete, which allows it to perform cryptanalysis and recover the secret key `rkey`.

2.1 Core Issue: No Abstraction for Security

To summarize, data-oblivious programming today is insecure and slow. It is insecure because of ISA-invisible microarchitecture-specific optimizations. It is slow because, out of fear of leaking privacy, programmers are forced into using only the simplest of instructions.

The paper sets out to address these issues by introducing new ISA-level abstractions for reasoning about security and enabling higher performance. A new ISA abstraction addresses the security problem by defining how instructions leak privacy *across all compliant microarchitectures*. It further enables higher performance by allowing data-oblivious programs to take advantage of *higher-performance instructions*, as long as those instructions are deemed safe by the ISA, and gives microarchitects the ability to optimize those instructions subject to the ISA-prescribed security policies.

3 Formal Definitions for Microarchitectural Side Channels

To start, the paper develops a security definition for microarchitectural side channel-free execution. There are two challenges. First, how to write the definition to account for any possible microarchitectural side channel. Second, how to write the definition so that it sheds insight on which instructions are "safe" from a microarchitectural side channel perspective.

To define privacy we adopt a trace-based indistinguishability-style definition inspired by the Oblivious RAM (ORAM) [7] literature. We consider a program λ which takes Public data x and Confidential data y as input. That program's execution trace, on a microarchitecture $\mu Arch$, i.e., "all the atoms in the universe that are perturbed as a result of running $\lambda(x,y)$ on $\mu Arch$," is denoted $\mu Arch(\lambda(x,y))$. The subset of this trace that the attacker can see (called the View) is denoted $\mathsf{View}(\mu Arch(\lambda(x,y)))$. For privacy, we require that the information in the View does not depend on Confidential information, i.e., that $\mathsf{View}(\mu Arch(\lambda(x,y))) \simeq \mathsf{View}(\mu Arch(\lambda(x,y')))$ for all Confidential data y and y'. In this setting, $\simeq$ informally means "equal, given the capabilities of any computationally-bounded adversary." For example, in ORAM schemes the View is the "memory access pattern" and ORAM seeks to make the memory access pattern indistinguishable as a function of Confidential data.

Next, we must define a View that captures any possible microarchitectural side channel that an arbitrary software-based attacker can monitor. This is non-trivial as the attacker can monitor many aspects of the program's execution. For example, its execution time, use of the cache, arithmetic units, etc. The paper makes a key observation that *all of these leakages can be modeled as Confidential data-dependent changes in the program's hardware resource usage over time*. For instance, both arithmetic units and cache sets are hardware resources and the fact that they are used at Confidential data-dependent times is the crux of the attacks.

Then, the question is how to determine whether a hardware resource is currently being "used" by a program. (Note that whether a hardware resource is being "used" is independent of the logic values currently stored in that structure.) For this, we rely on an explicit gate-level information flow abstraction similar to GLIFT [10]. Figure 1 shows an example using an ALU with operand-independent and then operand-dependent timing.

First assume a single-cycle ALU (Figure 1, Case 1). Suppose the input arrives and is stored in the input latches at the rising edge of cycle 1. Using terminology from information flow, we say the input latch is tainted in cycle 1. Now, regardless of the logic values of the input, the same latches are tainted in each cycle thereafter. That is, the output latches are tainted in cycle 2, etc. Because *which latches are tainted when* is independent of the operands, we say the single-cycle ALU does not form a microarchitectural side channel.

Next assume an ALU with operand-dependent timing (Figure 1, Case 2). For example, a multiply operation that takes 1 or 2 cycles, depending on whether an operand is 0. In this case, depending on the input, the output latch is either tainted in cycle 2 or cycle 3. Because *which latches are tainted when* is dependent on the operands, we say this ALU can form a microarchitectural side channel.

Putting everything together, we model the processor as a state machine composed of combinational logic and latches.[2] The subset of latches that store the Confidential input are denoted tainted at the start. Then, the View outputs a trace that indicates which subset of latches are tainted in each cycle. That is, hardware resource usage as a function of time. If the microarchitecture ensures that the View is independent of Confidential data, the microarchitecture does not leak privacy. Conversely, if the definition is not satisfied, we can pinpoint which instruction caused the problem by looking at where the Views diverged. (To note, the paper defines taint propagation in a non-standard way to model only explicit information flows. This prevents taint explosion, which would render the definition not useful. Implicit flows are dealt with by quantifying over all y'.)

4 Principles for OISA Design

The design principles for OISAs are two-fold. First, the OISA should expose security guarantees in a microarhictecture-independent way. That is, programs written using an OISA should maintain the same security guarantees across all OISA-enabled microarchitectures. Second, OISAs should not preclude modern hardware performance optimizations, except when those optimizations have a chance to leak privacy.

To address these goals, the OISA abstraction proposed in the paper has two parts. First, the OISA labels *data* to be

[2] W.l.o.g., we treat any state element (flip-flop, SRAM cell, etc.) as a latch.

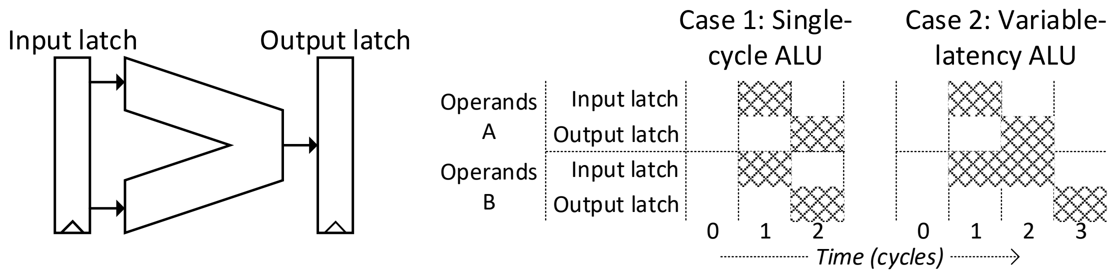


Figure 1: Changes in resource usage, as a function of data, create microarchitectural side channels. If a latch is shaded in a given clock cycle, that means there is (explicit) information flow from the operands to that latch in that cycle. Assume operands A and B are two sets of distinct data values, meant to induce different ALU timings.

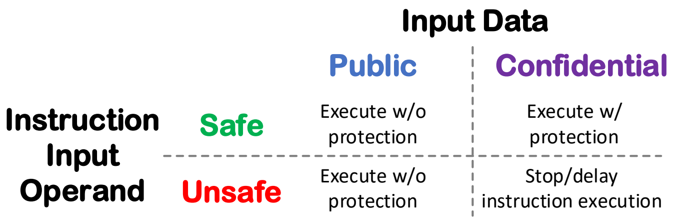
Confidential/Public, to capture whether that data is a function of user secrets (i.e., the sensitive program inputs from Section 3). Second, the OISA specifies, for each instruction, whether each *instruction operand* is Safe or Unsafe.

Finally, compliant microarchitectures must monitor and take different actions based on what data is consumed by what instruction operands at runtime. Specifically, hardware must enforce the following rules (shown in Figure 2):

- ($Confidential \nrightarrow Unsafe$) When Confidential data is presented to an Unsafe operand: the hardware must delay that instruction's execution until it is non-speculative. If the rule still applies when the instruction is non-speculative, the program terminates with a fault (as continuing would constitute an information leak).

- ($Confidential \rightarrow Safe$) When Confidential data is sent to a Safe operand: the hardware designer must add mechanisms to enforce the security definition given that instruction's execution (Section 3), for a specified View. For example, by disabling performance optimizations, scrubbing side effects and masking exceptions that occur as a function of Confidential operands.

- ($Public \rightarrow Safe/Unsafe$) When Public data is sent to Safe or Unsafe operands, no special treatment is needed and execution can proceed without protection.



		Input Data	
		Public	Confidential
Instruction Input Operand	Safe	Execute w/o protection	Execute w/ protection
	Unsafe	Execute w/o protection	Stop/delay instruction execution

Figure 2: Protection policies, checked before each instruction executes.

Despite these rules' simplicity, they provide both security and efficiency benefits. As we will see in Section 7, they provide a uniform handling for both traditional- and speculative execution-based attacks [8]. Case in point, the only mention of speculation is a detail in the rule for $Confidential \nrightarrow Unsafe$, where we say such an information flow delays the instruction's execution until it is non-speculative. This removes false-positive violations due to benign miss-speculations. At the same time, the rules enable blocking attacks with low overhead. Case in point, the rules encode some intuitive optimizations such as "Public data does not need protection" and "it is safe to compute on Confidential data with Safe instructions." The only situation where instruction execution is impeded is if Confidential data is consumed by an Unsafe operand.[3]

Key Idea: Abstract security policies facilitate programming simplicity, implementation flexibility and performance optimizations. Specifying instruction operand security policy *abstractly*, i.e., as Safe/Unsafe, 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 security vis. Section 3 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., [2]), hardware partitioning (e.g., [11]) or cryptographic techniques (e.g., [9])—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.

Information flow policy and implementation. The above OISA framework describes a relatively simple security lattice (akin to $\{High, Low\}$) [4], policy ($High \nrightarrow Low$) and information flow propagation rule (as written, data should be marked Confidential if its value is interferrent with the program's Confidential inputs, which implies a taint algebra akin to GLIFT [10]). This reflects the paper's goal: to provide a comprehensive, but simple, privacy guarantee for data-oblivious programming, while granting implementation flexibility to trade off design cost and performance. Given other use cases these parameters, and how they are concretely enforced by an implementation, can be changed for a family of OISAs, a particular OISA, or a microarchitecture that implements that OISA. For example, to support richer security lattices [6].

5 Design of a Concrete OISA

With the principles in Section 4, we now propose a baseline concrete OISA that can be easily implemented on top of common existing ISAs (e.g. x86, ARM, RISC-V).

[3]This principle directly inspired our follow-on work to block, specifically, speculative execution attacks [12].

Base Data Oblivious ISA Extension:

Arithmetic	operand$_1$ (Safe),	operand$_2$ (Safe)
Classify	operand (Safe)	
Declassify [serialized]	operand (Safe)	
Branch	predicate (Unsafe),	target (Unsafe)
Load	addr (Unsafe)	
Store	addr (Unsafe),	data (Safe)

Oblivious Memory Extension:

Oblivious Load	addr (Safe)	
Oblivious Store	addr (Safe),	data (Safe)

Figure 3: Data-Oblivious ISA policy when data is passed to an instruction operand.

Figure 3 highlights the instructions included in the OISA, and which operands are Safe/Unsafe. Programmers that write data-oblivious code will recognize this as a formalization of the guidelines used by data-oblivious programs today (Section 2). *Arithmetic* represents all binary arithmetic operations with Safe input operands. *Classify* promotes its operand from Public to Confidential. *Declassify* is opposite to Classify, which demotes its operand from Confidential to Public. *Branch* performs a conditional branch, but is only allowed to specify a Public destination or check a Public predicate. *Load* and *Store* are only permitted to take Public addresses. An important detail is that since Declassify has the potential to make previously-protected data vulnerable, the OISA requires that the Declassify instruction be verified as corresponding to program semantics. For example, on a speculative microarchitecture this would entail delaying such an instruction until it is non-speculative.

Extension: Memory obliviousness via Safe-address loads. A common bottleneck in existing data-oblivious code is the inability to use Confidential data as a load address. Therefore, we propose a new set of instructions (an oblivious-memory extension) that enable *memory-oblivious* computation [9]. Given the OISA design principles, enabling memory-oblivious instructions is conceptually simple. Instead of emulating memory obliviousness with dummy memory operations, we designate new load/store instructions whose address operand is *Safe*. This gives hardware designers the ability to build secure and efficient implementations, e.g., using partitioning [11] or Oblivious RAM [7], for that specific operation.

Load instructions with Safe address operands are just one example of how to accelerate secure computation with an OISA, and the paper leaves extending our concrete OISA with additional Safe instructions as future work. A key insight that motivates this direction is that many data-oblivious codes share common kernels (e.g., sorting) that become performance bottlenecks because the only available Safe operations are simple instructions. By encapsulating these larger operations into new instructions with Safe operands, a future OISA can potentially achieve constant factor or even asymptotic performance improvements. For example, a sort implemented data obliviously with simple Safe instructions may cost $O(n * \log^2 n)$ operations if implemented as a bitonic sort. On the other hand, if sort is specified as a single Safe 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.

6 Hardware Prototype on RISC-V BOOM

We prototype all hardware changes needed to support our OISA on top of the RISC-V BOOM processor (for "Berkeley Out-of-Order Machine") [3]. 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.

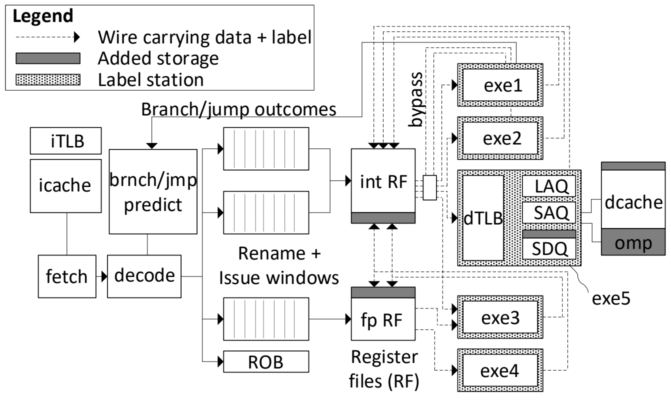


Figure 4: Microarchitectural changes needed to support the OISA from Section 5 (including the oblivious-memory extension, denoted 'omp'). Label stations check and enforce the transition rules from Figure 2.

Microarchitectural changes to support the OISA are shown in Figure 4. The main changes are logic at instruction issue/execute to enforce the rules from Section 4, storage/logic to implement the oblivious-memory extension and logic to track and denote data as Confidential/Public. For the latter, we implement a hardware information flow tracking mechanism similar to hardware dynamic information flow tracking, but capable of checking and updating whether data is Confidential/Public (the data's label) at any stage in the pipeline.

7 Formal Analysis

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. Through this model, we prove that the OISA provides a basis to satisfy strong security definitions such as those we defined in Section 3. Our security analysis is general, and applies given any implementation of several important processor structures (e.g., it models the branch predictor as an arbitrary function that takes previous branch resolutions as input).

Importantly, we are able to prove security *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.

Security intuition. Informally, to argue security we need to show that

(a) each instruction's resource usage/side effects is independent of Confidential data, and,

(b) the sequence of instructions that are executed, i.e., the processor program counter (PC), is independent of Confidential data.[4]

(a) follows by definition by applying the rules in Figure 2 to each instruction as it executes. A more subtle point is that

[4]Similar requirements on "not tainting" the PC also govern prior work [10].

(b) also follows from applying the same rules. To see why, first consider a simple un-pipelined, in-order processor with no speculation. In this case, it is clear (b) holds because the only instruction type from Figure 3 that changes the PC as a function of data is a branch, and the OISA requires that the branch predicate and target be Public data. What happens when we consider more advanced pipelines, e.g., with prediction and speculation? In that case, *microarchitectural state* outside of program semantics, e.g., the branch predictor state, influences the PC. To extend our security argument to these machines, we must extend what we mean by "resource usage/side effects" to include these structures. Then, using induction one can show that if (a) and (b) hold up to fetching the i-th instruction, the branch predictor state when fetching the $i+1$-th instruction is independent of Confidential data, and security follows.[5]

Example: Security against speculative execution attacks. The above reasoning shows how OISAs enable security against both non-speculative and speculative attacks. Consider the example speculative execution attack on data-oblivious decryption from Section 2. This attack does not go through when using an OISA by invoking conditions (a) and (b) above. That the branch predictor miss-speculates and executes `leak()` prematurely is not a function of Confidential data due to (b). Further, when `leak()` executes, it will be unconditionally stopped by the $\textit{Confidential data} \not\rightarrow \textit{Unsafe operands}$ rule due to (a). Extending the analysis to attacks like Spectre [8], where the branch predictor is intentionally miss-trained, reuses the same logic. That is, if (a) and (b) hold, then the attacker's strategy for how to miss-train the branch predictor cannot be a function of Confidential data because the program has not leaked Confidential data up to this point. In that case, intentional miss-training looks the same to the analysis as accidental miss-speculation, and security follows.

8 Evaluation

We evaluate OISAs 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, the OISA 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 the OISA speeds up constant time AES by $4.4\times$ and the memory oblivious ZeroTrace [9] 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`.

9 Discussion and Future Directions

OISAs can be extended in numerous directions, in particular as a way to compose existing hardware/software defensive mechanisms and as a novel backend for the Data-Oblivious Stack.

9.1 Simplifying and Composing the Hardware TCB

A major impediment to progress is that many hardware structures create side channels, and it isn't clear whether the program is "secure" until all channels are blocked. OISAs dramatically simplify this problem, enabling a new incremental methodology for designing secure hardware and software.

In their most basic deployment, an OISA might opt to only support very basic Safe instructions (e.g., bitwise operations and basic arithmetic). Such an OISA can likely be implemented with minimal changes to modern processors and already improves the state of security today. For example, by increasing our confidence that conservatively-written codes such as constant-time codes are really "constant time."

Beyond this basic deployment, however, OISAs provide a way for computer architects to plug-and-play their high-performance "point" defenses in a compositional way. For example, a Safe load can be implemented by previously-proposed partitioned or randomized cache architectures [11]. Importantly, architects need only worry about how to implement the Safe load. The generic OISA rules, e.g., $\textit{Confidential data} \not\rightarrow \textit{Unsafe operands}$, take care of the rest.

9.2 Composing with the Data-Oblivious Stack

Beyond writing side channel-free code for today's hardware, there is a rich literature in the applied cryptography community—ranging from algorithm/data structure design to language and compiler support—for performing secure multi-party and encrypted computation.

We make a key observation that the underlying programming abstraction assumed for those works is the same abstraction provided by an OISA. For example, a homomorphic encryption operation is akin to a Safe instruction, just using a different implementation suitable for a different threat model. This enables a new, large-scale research agenda to port insights and advances made in the applied cryptography community to/from the microarchitectural side channel community. For example, we can enable high-level programming abstractions for writing OISA-secure code by adding a new OISA backend to existing data-oblivious compiler frameworks. At the same time, the notion of Safe instructions provides a new theory to explore in applied cryptography. In particular, algorithm design in encrypted computation assumes only extremely simple Safe operations (e.g., bit add or multiply). With an OISA, however, we can choose which operations support Safe operands, and co-design algorithms with this in mind to improve performance.

[5]This idea to keep the predictors a function of Public data directly inspired the mechanism to block "implicit channels" in our follow-on work [12].

REFERENCES

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

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *IEEE S&P'15*.

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

[4] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, May 1976.

[5] 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*.

[6] 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*.

[7] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, May 1996.

[8] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Man-

gard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *IEEE S&P’19*.

[9] S. Sasy, S. Gorbunov, and C. W. Fletcher, “Zerotrace : Oblivious memory primitives from intel sgx,” in *NDSS’18*.

[10] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *ASPLOS’09*, 2009.

[11] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *ISCA’07*, 2007.

[12] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. Fletcher, “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data,” in *MICRO’19*.