Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data

Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel[†], Adam Morrison[‡], David Kohlbrenner[§], Christopher W. Fletcher
University of Illinois at Urbana-Champaign, [†]Stanford University, [‡]Tel Aviv University, [§]University of Washington
{josers2, pshome2, ndnayak2, cwfletch}@illinois.edu, trippel@stanford.edu, mad@cs.tau.ac.il, dkohlbre@cs.washington.edu

***Abstract*—Microarchitectural attacks have plunged Computer Architecture into a security crisis. Yet, as the slowing of Moore's law justifies the use of ever more exotic microarchitecture, it is likely we have only seen the tip of the iceberg.**

To better anticipate this security crisis, this paper performs a systematic security-centric analysis of the Computer Architecture literature. Our rationale is that when implementing current and future processors, microarchitects will (quite reasonably) look to previously-proposed ideas. Our study uncovers seven classes of microarchitectural optimization with novel security implications, proposes a conceptual framework through which to study them and demonstrates several proofs-of-concept to show their efficacy. The optimizations we study range from those that leak as much privacy as Spectre/Meltdown (but without exploiting speculative execution) to those that otherwise undermine security-critical programs in a variety of ways. Many have storied histories—ranging from industry patents to media/3rd party speculation regarding current implementation status to recent renewed interest in the academic community. This paper's goal is to perform an early (hopefully not too late) analysis to inform their development moving forward.

I. Introduction

Computer Architecture is grappling with a security crisis. Starting with Spectre and Meltdown [1, 2], there has been a tidal wave of new security vulnerabilities attributed to *microarchitecture* (e.g., [3–5]), which has since breathed new life into the rich sub-area of *microarchitectural attacks* [6].

This paper's thesis is that the slowing of Moore's law will significantly exacerbate this security problem. Without transistor scaling, it follows that microarchitects will employ ever more exotic microarchitectural optimizations to improve performance. If the past is any indicator of the future, it stands to reason that these optimizations might have novel—perhaps even devastating—security implications. Further, beyond "simply" impacting future processors, it is likely the case that some such microarchitecture already exists in current hardware, creating latent, not-yet-discovered vulnerabilities. Examining the deluge of exploits that have come out in the last three years, it is clear that processor complexity has outpaced attacker bandwidth to find hardware zero days.

To better anticipate this security crisis, the goal of this paper is to perform a systematic study of the Computer Architecture literature, through a security lens. Our rationale is that when implementing current and future processors, microarchitects will (quite reasonably) look to previously-proposed ideas. Quoting the website of the recent value predictor competition, "value prediction has regained interest in an era where Moore's Law is fading and Dennard Scaling is gone" [7].

Motivating example: data memory-dependent prefetchers leak as much privacy as Spectre/Meltdown. As a motivating example, consider prefetching. While the community is well-aware of the security implications of "traditional" stride and software prefetching [8, 9], this is only the tip of the iceberg in the prefetcher literature. In particular, there is significant literature on *data memory-dependent prefetchers*, i.e., prefetchers that take into account the *contents of data memory directly* [10–16]. Such prefetchers are effective in cases where stride prefetchers fail, e.g., in applications dominated by indirections or "pointer chasing."

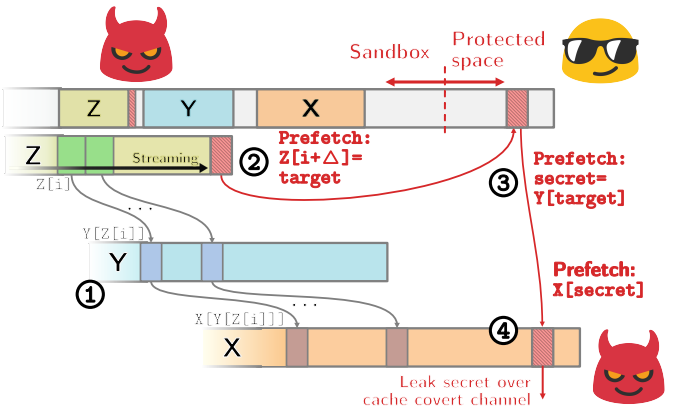

Fig. 1: Indirect-memory prefetcher leaking all of program memory (forming a *universal read gadget* [17]) in the sandbox setting. The attacker can choose which word of private data it wants to leak by setting the value `target` relative to the base address of array Y. The victim's sunglasses represent the sandbox's software-level memory-safety checks.

Such prefetchers are prime implementation candidates as they tackle the critical and ever-increasing memory bottleneck problem. Take for example the "indirect-memory prefetcher" [13] (recently patented by Intel [18]). The multi-level design [13] (Listing 3) tries to detect programs of the form $\texttt{for}(i = 0...N)\ X[Y[Z[i]]]$ and prefetch the cache line corresponding to $\&X[Y[Z[i + \triangle]]]$, which requires the prefetcher to sequentially prefetch (make cache accesses for) $z = Z[i + \triangle]$, followed by $y = Y[z]$, followed by $X[y]$.

Now consider the above prefetcher in an adversarial setting, as shown in Figure 1. Suppose the attacker wants to read

memory outside of a software sandbox (e.g., in a browser [19] or eBPF [20] in the Linux kernel, the latter of which was also targeted in the original Spectre disclosure [21]). ① In that case, the attacker can run the now-malicious program $\mathsf{for(i = 0...N)\ X[Y[Z[i]]]}$, or another like it, inside the sandbox to activate the prefetcher. ② By controlling the contents of memory immediately after the bounds of Z, the attacker can trick the prefetcher into reading attacker-controlled data $\mathsf{z = Z[i + \triangle]}$ out of bounds of Z. ③ This implies the attacker can read an attacker-chosen private value $\mathsf{y = Y[z]}$, i.e., out of bounds of Y, and ④ finally leak/transmit that value over a traditional cache covert channel (e.g., Prime+Probe [22]) vis. the final prefetch for $\mathsf{\&X[y]}$. We provide more details on this attack in Section V-B.

By repeating the attack for different z, the attacker can leak all memory outside the sandbox, i.e., form a *universal read gadget* [17]. *To our knowledge, this is the first microarchitectural optimization outside of speculative execution that can be used to build universal read gadgets with realistic software assumptions*—suggesting such devastating privacy leakage could be a more systemic problem than previously believed.

Security analysis of broad classes of microarchitectural optimizations. Beyond data memory-dependent prefetchers, this paper studies the security implications of **six** additional microarchitectural optimization classes—covering computer architecture principles ranging from value locality to compressibility to prediction to prefetching to more generally hardware "fast paths". In total, we analyze:

1) computation simplification (e.g., [23])
2) pipeline compression (e.g., [24])
3) silent stores (e.g., [25])
4) computation reuse (e.g., [26])
5) value prediction (e.g., [27])
6) register-file compression (e.g., [28])
7) data memory-dependent prefetchers (e.g., [13])

We additionally provide discussion on continuous/trace-based optimization (e.g., [29]).

There is significant evidence to suggest many of the above optimization classes will make it to commercial products in the future, if they have not already. For example, third-party analysis provides evidence that a limited form of silent stores is actually implemented on Intel machines today [30], and support for silent stores is explicitly mentioned in the RISC-V memory-consistency model documentation [31]. As mentioned above, value prediction has recently seen a resurgence of interest, with all-year competitions held to spur innovation [7] and test-of-time awards given to the originating proposals [32]. Also mentioned above, the data memory-dependent prefetcher used in our motivating example was recently patented by Intel [18]. Further, many other industry patents have been issued for the other optimization classes over the years to a variety of companies (e.g., [33–36]). Finally, limited forms of computation simplification and pipeline compression have indeed been implemented—leading to demonstrated attacks (see [37] and [38], respectively).[1]

Focus: microarchitecture with novel privacy implications. Perhaps most importantly, we chose to study the above seven optimizations because they can leak data beyond what is possible by prior attacks, and/or via previously-unstudied microarchitectural mechanisms. Studying such "novel" microarchitecture is important, as resulting attacks will most likely evade current defenses. For example, while there have been many defenses proposed to block the universal read gadget caused by Spectre (e.g., [39–41]), such mechanisms are insufficient for blocking the universal read gadget caused by a data memory-dependent prefetcher. Further, all optimizations we study break the widely-deployed software defense paradigm known as constant-time programming (e.g., [42–46]).

Contributions. To summarize our contributions:

1) We perform the first comprehensive security-centric study of previously-proposed microarchitectural optimizations. In totality, we analyze the following optimization classes: data memory-dependent prefetching, silent stores, computation reuse, computation simplification, pipeline compression, register-file compression, value prediction and continuous/trace-based optimization—based on their novel security implications.
2) We propose a conceptual framework for reasoning about leakage through the above microarchitectural optimizations. The framework precisely describes what information leakage is possible through each, in terms of interactions between in-flight instructions, microarchitectural state and architectural state—in the context of both passive and active attackers.
3) We provide a proof-of-concept implementation of silent stores in an architectural simulator, develop a technique to amplify timing differences stemming from it and use that technique to break constant-time cryptography. We also provide a detailed analysis of the data memory-dependent prefetcher exploit, in the context of the eBPF sandbox.

Open source. We have open-sourced proof-of-concept infrastructure related to the project here: https://github.com/FPSG-UIUC/Pandora.

II. UArch Attack Primer and Threat Model

Microarchitectural attacks are a class of side/covert-channel attack that enable communication and privacy leakage due to processor microarchitecture. In the *side-channel* setting, a victim program runs and an attacker program that observes what hardware resources the victim uses, and when, infers the victim's private data. In the *covert-channel* setting, two attacker programs attempt to communicate with each other through hardware resource usage intentionally. In both of the above, attacks can be modeled as a *transmitter* (i.e., victim/sender in the side/covert channel settings, respectively)

[1] Although, as we will show, these two optimization classes are more general than what is currently known to be implemented, and thus warrant further investigation.

modulating *channel* (hardware resource), which is then interpreted by a *receiver* (typically another thread) [47]. In many cases, the transmitter is a specific instruction that modulates the channel as a function of its operands. We refer to these as *transmit instructions* for short [39]. The receiver measures the channel using sources of non-determinism available to it, that are a function of microarchitectural resource usage. These are usually explicit timer instructions or implicit clocks constructed through shared memory [37, 48].

1) **Channels and privacy leakage:** Over the years, many hardware structures have been exploited as channels, including the cache [22, 49, 50, 50, 51], TLBs [52], page tables [53, 54], DRAM [55], branch predictors [56, 57] and others [8, 9, 37, 38, 44, 58–60]. Any of the above can be used to construct a covert channel.

A more subtle question, central to this paper, is *what program data do each of the above leak in the side channel setting?* In this way of thinking, the above landscape changes significantly. For example, memory-related channels (e.g., through the cache, TLB, DRAM, etc.) only directly leak functions of the victim's memory access pattern—i.e., data passed as the address operand of a load or store instruction. Likewise, branch prediction and arithmetic unit port contention [59] channels leak the victim's control flow—i.e., data passed as branch predicate/target operands. In this view, speculative execution attacks such as Spectre [1] are novel, as they enable incorrect (transient) sequences of instructions to steer previously un-reachable data to transmitters.

2) **Active attacks and performing multiple experiments/replays:** Microarchitecture attacks are typically *active attacks*, i.e., where the attacker explicitly primes (or preconditions [61]) system state to amplify leakage. For example, in Spectre variant 1 [1]—`if(i < N) Y[X[i]];`—the attacker controls the index `i` to specify which private value it wishes to leak. Likewise in the Safecracker attack on compressed caches [4], the attacker tries to co-locate its own data next to private data to induce compression as a function of the private data. Attackers can also precondition microarchitectural state, e.g., in Prime+Probe attacks [22].

It is also normally assumed that attackers can perform many experiments/replays [1, 4, 57, 62]. Each experiment can use the same preconditioning (to reduce noise) or a different preconditioning (to amplify leakage). For example, in the case of Spectre, the attacker can change `i` over different experiments to leak information about multiple private data values. Orthogonally, in the case of Safecracker, the attacker can change what data is co-located next to a specific private data value to incrementally reduce uncertainty about that private data (also called *differential analysis* [63]).

3) **Attack scenarios:** Our proofs-of-concept (Section V) assume two popular attack scenarios called *cloud* and *sandbox*. The scenario influences how the attacker performs active attacks, where the receiver can run and what the receiver can measure.

In the cloud setting, the attacker runs on a remote device and interacts with the victim program through some well-defined interface. Two common examples are interacting with remote web servers through network endpoints or interacting with SGX enclaves through the SGX ECALL interface [64]. Depending on the setting, the attacker has user- or supervisor-level privileges and the receiver might be the remote attacker itself (in which case, it can monitor victim runtime over the network [65]), or run concurrent to the victim on the same processor (e.g., as an SMT sibling or on another physical core [6, 66, 67], or as the kernel [53, 68, 69]).

In the sandbox setting, the attacker is able to run its own code in the victim's address space. Two examples are Linux eBPF [20] and web browser sandboxes (e.g., NaCl [19]). For example, eBPF enables the kernel to "safely" run user-specified code in kernel space when specific OS events, such as system calls, occur. The kernel (eBPF sandbox creator) takes steps to ensure that such code cannot read/write memory outside the sandbox, e.g., by checking that it is memory safe. The attacker's goal is to circumvent these protections. In this setting, the receiver is typically the attacker-controlled program in the sandbox.

4) **Out of scope: physical and energy-related attacks:** In both of the above settings, the attacker is software-based and monitors microarchitectural resource usage. We treat analog channels, that require physical equipment to monitor, e.g., power draw [63] or EM emissions [70], as out of scope. We also do not consider interactions between microarchitectural state and clock frequency/power-states (e.g., DVFS effects) [71, 72], although this is important future work.

III. Focus: UArch with Novel Privacy Implications

Every microarchitectural optimization inherently creates program data-dependent resource usage, and therefore potentially reveals information to attackers who can monitor that usage. This paper's goal is to look at a meaningful subset of these optimizations, namely those that leak program data that was not known to be at risk before.

For different microarchitecture variants, we show what data is at risk in Table I. Data can either be leaked *in use* (i.e., when passed to or returned by a transmit instruction) or *at rest* (i.e., where a transmitter activates based on architectural state directly). For data-in-use leakage, we further break down which instruction(s) causes the leak and whether operand or result values leak. For data-at-rest leakage, we break down what architectural state (the register file or memory) leaks. In both cases, we assume the victim program is run in the presence of the software-based attacker (i.e., receiver) described in Section II-3.[2]

This view provides useful security-related information for both attackers and defenders. For example, the widely-deployed mitigation *constant-time programming* (e.g., [42–46]) attempts to block attacks by ensuring that no private data is computed on by transmit instructions, and speculative execution attacks [1] use transient execution to pass private

[2]This receiver is equivalent to the 'idealized' BitCycle attacker used in [45] that can monitor hardware resource usage at flip-flop and clock-cycle granularity.

data to transmit instructions [39, 47]. In both of the above, the first step is to identify what are the transmitters latent in a given microarchitecture.

TABLE I: Leakage landscape. S, U, U′ denote Safe, Unsafe and Unsafe (but leaking a different function of the data than U, or under different assumptions), respectively. ‡ indicates that this data may be Unsafe if the victim program contains an appropriate speculative execution gadget [1]. For arithmetic operations, operands (as opposed to the result) are considered U if the transmitter can be fully specified by a partition on the operand assignment space of in-flight dynamic instance(s) of the given instruction type (Section IV). Representative citations are given for each data type that the Baseline microarchitecture leaks.

Acronym meanings: CS = computation simplification, PC = pipeline compression, SS = silent stores, CR = computation reuse, VP = value prediction, RFC = reg-file compression, DMP = data memory-dependent prefetching.

		Baseline	CS (IV-B1)	PC (IV-B2)	SS (IV-C1)	CR (IV-C2)	VP (IV-C3)	RFC (IV-D1)	DMP (IV-D2)
Data in use (operands/result of a transmit instruction)									
Operands	Int simple ops	S	U	U	-	U	-	-	-
	Int mul	S	U	U	-	U	-	-	-
	Int div	U [44]	U′	U′	-	U′	-	-	-
	FP ops	U [37]	U′	-	-	U′	-	-	-
Result	Int simple ops	S	-	-	-	-	U	U	-
	Int mul	S	-	-	-	-	U	U	-
	Int div	S	-	-	-	-	U	U	-
	FP ops	S	-	-	-	-	U	U	-
Addr	Load	U [49]	-	-	-	-	-	-	-
	Store	U [49]	-	-	-	-	-	-	-
Data	Load	S	-	-	-	-	U	-	-
	Store	S	-	-	U	-	-	-	-
	Control flow	U [56]	-	-	-	-	-	-	-
Data at rest									
	Register file	S ‡	-	U	-	-	-	U	-
	Data memory	S ‡	-	-	U	-	-	-	U

In the table, we analyze leakage relative to a "Baseline" architecture. The baseline represents a typical commercial server processor (out-of-order, speculative, multicore) based on known attacks today (Section II). Each column represents the same baseline plus the specified microarchitectural optimization (that we study in this paper). For example, the column "SS" indicates the baseline architecture with silent stores added. U/S indicates that data does/does not leak (is "Unsafe"/"Safe") and each column shows the difference between the baseline and the baseline plus each optimization. For example: given silent stores, store data transitions from S → U because silent stores create a transmitter as a function of in-flight store data. '-' indicates no change in safety relative to the baseline.

Goal 1: Security analysis of optimizations that expand leakage relative to Baseline (Table I). As shown in Table I, each optimization we study (the seven rightmost columns) increases the scope of what program data can be leaked relative to what is known today.[3] That is, all optimizations we study break current constant-time programming, reduce the assumptions needed to launch speculative execution attacks or otherwise leak previously-considered-safe data. A meta takeaway is that, if one considers the union of all optimizations we study, no instruction operand/result (or data at rest) is safe.

Goal 2: Reason precisely about privacy leakage. Beyond finding optimizations that simply endanger previously safe data, we are interested in understanding *precisely what function of the private data leaks and under what circumstances*. This is a subtle but important point. For example, in the cache-based Prime+Probe attack, only the address bits used to lookup the cache set are unsafe [22], and only if the attacker primes the cache set beforehand (Section II-2). All of the optimizations we study have similar subtleties. In the next section, we provide a framework for precisely specifying what data leaks and under what circumstances. In Table I, we indicate that an optimization leaks a different function of already Unsafe (U) data with the symbol U′.

IV. Characterizing Novel Information Leakage through Microarchitecture

In this section, we analyze the security implications of the microarchitectural optimizations from Table I.

A. Microarchitectural Leakage Descriptors: Precisely describing privacy leakage

Before we begin, we develop a conceptual framework for concisely, yet precisely, describing what privacy can leak through microarchitecture.

Microarchitectural optimizations trigger hardware resource usage changes due to interactions between in-flight dynamic instructions, ISA-invisible persistent microarchitectural state (e.g., predictors, caches) and/or ISA-visible persistent architectural state (e.g., the register file, data memory). We describe which such interactions result in which distinct observable outcomes (i.e., distinct modulations of hardware resources that form a channel; Section II) using what we call *microarchitectural leakage descriptors (MLDs)*. An MLD for a given microarchitectural optimization is a stateless function that specifies:

- The inputs needed to describe the optimization's functional behavior. Each input can have one of three types: Inst, Uarch or Arch for dynamic instruction, persistent microarchitectural state and architectural state, respectively.
- A many-to-one mapping from assignments of these inputs to distinct observable outcomes.

Given a concrete assignment to the inputs, the MLD returns a unique id (natural number) indicating which distinct observable outcome the input assignment corresponded to. This mapping partitions the MLD's input assignment space.

1) **Example MLDs:** We show three example MLDs in Figure 2, for several microarchitectural structures covered in prior work. We use Python-like notation and assume that MLD arguments have convenience fields, e.g., Inst has a PC, opcode,

[3] Speculative execution attacks (e.g., [1]) are not explicitly shown in Table I because, as noted above, they use transient execution to steer private data towards transmit instructions, whereas the table's goal is to articulate what instructions are transmit instructions.

```
// Example 1: single-cycle addition.
mld single_cycle_alu(Inst i1): return 0

// Example 2: zero-skip multiply.
mld zero_skip_mul(Inst i1): return ∨_i i1.arg.v_i == 0

// Example 3: cache w/o shared memory and w/ a
random replacement policy. Assume i1 is a load,
addr is the load address and cache is the current
cache state.
mld cache_rand(Inst i1, Uarch cache):
  return set(i1.addr.v) + 1 if i1.addr.v ∉ cache else 0
```

Fig. 2: Example microarchitectural leakage descriptors (MLDs) for optimizations covered in prior work. set(.) returns the cache set that its argument (an address) will map to. ∨ denotes logical OR.

etc. When referring to operand/result values vs. register IDs, we add a suffix .v and .id, respectively.

In Example 1, a single-cycle ALU "takes as input" [the operands of] a single dynamic instruction (Inst) and produces the result one cycle later, given any possible assignment to the operands. Thus, the MLD unconditionally outputs 0, indicating the ALU does not form a transmitter, i.e., is Safe (Section III).

In Example 2, a multiplier takes a single cycle to execute if neither operand is 0, but skips (takes 0 cycles) if either operand is 0. This creates two distinguishable outcomes, in the form of timing differences (similar to [37, 38]), and the MLD definition indicates which occurs as a function of the instruction i1's operands' values $i1.arg.v_i$ for $i = [0, 1]$.

In Example 3, a cache is accessed through a memory instruction i1. Assuming no shared memory and a random replacement policy, this creates (num_cache_sets + 1) distinguishable outcomes: one for each possible setting of i1's address' set bits (if the cache line corresponding to the address is un-cached) and one additional outcome (in the event of a cache hit). Modeling caches with other replacement policies is analogous; the MLD mapping must simply be extended to take into account extra information transmitted via replacement state.

2) **Public vs. private vs. attacker-controlled data:** In describing leakage, it is important to understand what data in the MLD's input is public, private and attacker controlled.[4] This determines to what extent an attacker is able to influence preconditioning (Section II-2). For example, in the zero-skip multiply (Example 2, above), if both operands are private, the attacker learns whether one or more of the inputs equals 0. If one operand is public and the other private, the attacker learns whether the private operand is 0 if the public operand is non-zero, but learns no information if the public operand is already 0. (If the public operand is 0, that the skip occurs is a purely a function of public information [73].) If one operand is attacker controlled and the other is private, and the attacker sets the attacker-controlled operand to a non-zero value, the multiply leaks precisely whether the other (private) operand is non-zero. Whether data is public, private or attacker controlled, as well as what preconditioning is possible for attacker-controlled data depends on factors such as the victim program and microarchitectural state.

3) **Estimating channel capacity:** Finally, we note that MLD specifications provide information about an optimization's channel capacity. That is, if S denotes the partition specified by the MLD, $\log_2 |S|$ denotes an upper bound on the number of bits that can be encoded into the channel at a given time.

4) **A preliminary characterization:** Using the MLD framework, the rest of the section describes each optimization class in detail. For each we discuss representative papers that, to the best of our findings, capture the design space and security implications of each optimization class. We organize the discussion based on Table II: based on whether an optimization creates a transmitter as a function of purely in-flight instruction(s) (Stateless instruction-centric; Section IV-B), in-flight instructions interacting with persistent microarchitectural/architectural state (Stateful instruction-centric; Section IV-C) or purely architectural state (Memory-centric; Section IV-D). We provide example MLDs for specific proposals of each optimization class in Figure 3, which we refer to throughout the text.

TABLE II: Optimization classification based on MLD signature.

	Instruction (Inst)			Memory (Arch) - IV-D
	Stateless - IV-B	Stateful - IV-C		
		Uarch	Arch	
Comp. simplification	√			
Pipeline compression	√			
Silent stores			√	
Computation reuse		√		
Value prediction		√		
Reg. file compression				√
D-memory prefetching				√

B. *Stateless Instruction-centric Optimizations*

We start by describing two optimization classes—computation simplification and pipeline compression—that act on only in-flight instruction(s).

1) **Computation simplification:** are techniques that simplify or eliminate instruction execution when operand values satisfy certain conditions [23, 77–82]. The zero-skip multiplier (Section IV-A1) is a well-known example, but the principle can be applied to everything from complex instructions (e.g., square root [77]) to the simplest integer operations (e.g., AND/OR/ADD/etc. [78, 80, 81]). Further, instruction execution can be simplified (e.g., divide converted to shift [77, 78]) or eliminated entirely (such as the zero-skip multiply in Section IV-A1). Finally, work saving can manifest in different forms and in different pipeline stages, e.g., skipping issue/execution [77, 78], register file reads [81], or both [79, 80]. Finally, proposals may require the presence of multiple in-flight instructions to trigger [23].

[4]This can be interpreted as the security lattice $\mathsf{L} \sqsubseteq \mathsf{C} \sqsubseteq \mathsf{H}$ where $\mathsf{L}, \mathsf{C}, \mathsf{H}$ denote public, attacker controlled and private data, respectively.

```
// Example 4 (Section IV-B): arithmetic unit operand
packing [24]. i1 and i2 must share the same execution unit
type.
mld operand_packing(Inst i1, Inst i2):
  return msb(i1.arg.v0) < 16 ∧ msb(i1.arg.v1) < 16 ∧
         msb(i2.arg.v0) < 16 ∧ msb(i2.arg.v1) < 16

// Example 5 (Section IV-C): silent stores [25]. i1 must be
a store.
mld silent_stores(Inst i1, Arch data_memory):
  return i1.data.v == data_memory[i1.addr.v]

// Example 6 (Section IV-C): dynamic instruction reuse, Sv
variant [74]. reuse_buffer is the PC-indexed memoization
table that records each memoized instruction's operand
values.
mld instruction_reuse(Inst i1, Uarch reuse_buffer):
  return  ∧i  i1.arg.vi == reuse_buffer[i1.pc][i]

// Example 7 (Section IV-C): value prediction [75].
prediction_table is the PC-indexed predictor table where
each entry contains a confidence conf (an unsigned
number) and a predicted value prediction.
mld v_prediction(Inst i1, Uarch prediction_table):
  return prediction_table[i1.pc].conf ||
         prediction_table[i1.pc].prediction == i1.dst.v

// Example 8 (Section IV-D): Register-file compression
given an N-entry register file, 0/1 variant [76] (which only
compresses if a given register's value is 0 or 1). Assume
register values are treated as unsigned values.
mld rf_compression(Arch register_file):
  return (register_file[N − 1] ≤ 1)|| . . . ||(register_file[0] ≤ 1)

// Example 9 (Section IV-D): 3-level indirect-memory
prefetching, for access pattern X[Y[Z[i]]] [13]. The
prefetcher state includes each array base, e.g., baseX
for &X[0], as well as the starting offset start for the
prefetch i + △.
mld im3l_prefetcher(Uarch imp,  Uarch cache,
                    Arch data_memory):
  s = imp.start // s = i + △
  z = data_memory[imp.baseZ+s] // z = Z[i + △]
  y = data_memory[imp.baseY+z] // y = Y[Z[i + △]]
  return cache_h(imp.baseZ+s, cache) ||
         cache_h(imp.baseY+z, cache) ||
         cache_h(imp.baseX+y, cache)
```

Fig. 3: Example microarchitectural leakage descriptors (MLDs) for optimization classes we study in Sections IV-B through IV-D. msb(.) returns the index of the most-significant ON bit in its argument. ∧ denotes logical AND. cache_h(.) refers to a cache MLD (e.g., cache_rand from Figure 2, Example 3), but taking an address as the first argument as opposed to a dynamic instruction. || denotes concatenation, or projection onto the natural numbers. That is: let $d_{N-1}, \ldots, d_0$ be functions of concrete assignments to MLD inputs and let $D_{N-1}, \ldots, D_0$ be the size of their domains. Then, $d_{N-1}||...||d_0 = \sum_{i=1}^{N-1}((\prod_{j=0}^{i-1} D_j) * d_i) + d_0$. Informally, this means the microarchitecture leaks information about each of $d_{N-1}, ..., d_0$ independently.

2) **Pipeline compression:** is a class of techniques that compress data as it travels through a processor pipeline, to reduce energy and/or improve throughput [24, 83–85]. The recurring idea is called *significance compression* [83]: that a word of data is effectively only as wide (bitwidth-wise) as its most-significant on-bit. Different schemes exploit this at different points in the pipeline and for different purposes. For example, to improve execution unit throughput by packing multiple narrow-width operands [24] or to improve bandwidth between pipeline stages or reads/writes to the register file [83–85]. Schemes further differ in granularity, e.g., whether they are capable of breaking operands into bits [84, 85], bytes [83], half-words [83].

3) **Security analysis:** The community has a nascent understanding of both of the above optimization classes as they have been implemented in limited form and have led to end-to-end attacks. For example, floating-point subnormal behavior [37] (computation simplification), digit-serial multiplication [38] (significance/pipeline compression) and division early exit [44] (taking ideas from both).

Beyond the above, we make the following observations. First, it is clear from the above work that the space of such optimizations is significantly broader than what has been seen so far. Pushed to the extreme, such optimizations can render even bitwise instructions, that are critical for constant-time programming, unsafe [78, 80, 81]. Second, many of the above are susceptible to active attacks, analogous to those on the zero-skip multiplier discussed in Section IV-A2.

Third, optimizations act not only on single instructions, but sometimes require interaction between multiple in-flight instructions [23, 24]. For example, consider the MLD for operand-packing execution units [24], shown in Figure 3, Example 4. This optimization improves execution unit throughput by packing two arithmetic operations into one if both operands for two pending instructions are sufficiently narrow width. This results in one of two observable outcomes, as shown by the MLD, and depends on both instructions being co-located to the same reservation station at the same time. This has implications for specific threat models. For example, a receiver in a sibling SMT thread can perform an active attack by setting its own instruction operands (i.e., i2's) such that the packing optimization occurs strictly as a function of a victim instruction's (i1's) operands.

C. Stateful Instruction-centric Optimizations

We now discuss three optimization classes that trigger due to interactions between in-flight instructions and either microarchitectural or architectural state.

1) **Silent stores:** are stores that do not change the contents of memory when they are performed [25]. Microarchitecture for silent stores tries to detect when this is the case, and skip performing such stores. Different proposals implement checking in different ways, in different pipeline stages. For example, at store retirement (comparing the in-flight store data with the contents of data memory) or in the load-store queue (comparing the in-flight store with an older in-flight store) [86] or speculatively in the decode stage [87].

2) **Computation reuse:** uses hardware to detect and eliminate redundant computations [26, 74, 77, 88–94]. At a high

level, these techniques use a key (identifying the computation) to lookup a hardware memoization table, using the result from the table and skipping the computation on a hit. This is non-speculative because table lookups cannot yield false positives, i.e., always yield correct results on hits.[5] Two relevant questions are: (1) what computations are memoized, (2) what key is used to lookup the table? For (1), proposals have covered a range of computations, from high-latency instructions [90], to potentially any arithmetic and memory instruction [74, 94], to subexpressions [92], to instruction traces [93]. For (2), some proposals lookup the memoization table by operand value, e.g., post register-file read [74, 77, 89, 90, 92–94]. Other proposals lookup the table based on operand ID, e.g., logical register [74, 88, 91]. The former typically achieves higher reuse [74] and is the focus of this section's discussion. We discuss the latter's implications on defenses in Section VI-A.

3) **Value prediction:** is a family of speculative optimizations intended to break instruction dependencies, thereby increasing instruction-level parallelism [27, 32, 75, 95–98]. At a high level, these schemes predict the result of instructions before they are computed. Similar to other prediction-based schemes, e.g., branch prediction, incorrect predictions lead to pipeline squashes. Further, resolved predictions update dynamic predictor state (if any) with information related to the resolved value. Prior work has proposed value predicting both just for loads [75, 96, 98] and also for broader classes of instructions [32, 97]. While different proposals cover a multitude of prediction heuristics—ranging from simple last-level and stride predictors to hybrid predictors [97, 99, 100]—nearly all are *threshold based*. This means they do not make predictions until those predictions are sufficiently high confidence, e.g., by maintaining counters that track the number of predictable values seen over time.

4) **Security analysis:** To our knowledge, prior work has not performed a security analysis of any of the above three optimization classes. Yet, third-party analysis indicates that a limited form of silent stores is implemented on Intel machines today [30], and support for silent stores is explicitly mentioned in the RISC-V documentation [31]. (We provide a proof-of-concept that demonstrates how silent stores renders constant-time code no longer constant time in Section V-A.) Likewise, value prediction has recently seen a resurgence of interest with all-year competitions held to spur innovation [7].

An important observation is that the above take a similar form from a privacy leakage standpoint. We show MLDs for silent stores, an implementation of computation reuse called "dynamic instruction reuse" [74] and a typical confidence-based value predictor [75] in Figure 3, Examples 5-7. The takeaway is that all three MLDs leak a function of whether an instruction operand/result value *equals* another value stored in either architectural or microarchitectural state.

[5]We note that while using a memoized result is non speculative, i.e., is always correct and will not cause a squash, the memoization table need not be cleared on a squash [26], implying that transient instructions can poison the table to transmit secrets through table state. This can be used as a covert channel in a speculative execution attack [1].

The above has important implications in the context of active replay attacks (Section II-2). Take for example silent stores (Figure 3, Example 5). The situation is similar for computation reuse and value prediction. If the contents of data_memory[i1.addr.v] (data memory at address i1.addr.v) are attacker controlled and the attacker can perform multiple experiments, the attacker can eventually learn i1.data.v precisely. That is, in each experiment the attacker can learn whether i1.data.v $== v$ for some v of its choosing, and vary v across experiments. Because these optimizations check for equality, the attacker can exponentially reduce the number of experiments needed to learn each value if it can perform checks with narrower-width v. For example, if v is a Word (Byte) then learning 32 (8) bits takes 2^{32} (2^8) tries in expectation, respectively.

Similar to attacks on branch predictors [39, 56, 57], these attacks are also symmetric. For example, in an attack through silent stores, i1.data.v might be private while data_memory[i1.addr.v] is attacker controlled, or vice versa. If i1.data.v is private, the attacker can learn the value of an in-flight instruction operand. If data_memory[i1.addr.v] is private, the attacker can learn data at rest (which may or may not have even been written by a store, e.g., considering other data write mechanisms such as DMA).

Finally, we expect all of the above to result in similar receivers relative to previous attacks on branch predictors. That is, receivers in attacks on branch predictors monitor program execution time, which is a function of the number of branch mispredictions (squashes). Similarly, silent stores improve throughput by freeing up the memory write port, computation reuse improves pipeline issue/execute stage throughput and value prediction causes squashes.

D. Memory-centric Optimizations

Finally, we discuss two classes of optimization that trigger as a function of data at rest, i.e., stored in the register file and data memory.

1) **Register-file compression:** exploits value locality to increase the effective number of physical registers in out-of-order pipelines, enabling greater instruction-level parallelism [28, 76, 101–103]. A typical approach is [76]: during the rename stage, allocate physical registers as usual. When an instruction produces a result, check if that result's value is already present in the register file. If so, return the physical register allocated to the result to the "free pool", which enables younger queued instructions to be renamed. Different schemes enable different values to be matched, e.g., any value [76, 101], or just common values such as 0/1 [76, 102]. Other schemes exploit different types of compressibility, e.g., by applying significance compression to pack narrow-width values [103], or by exploiting "value similarity" [28].

2) **Data memory-dependent prefetchers:** are a class of prefetchers that take program data memory contents as inputs *directly* (as opposed to just program memory addresses) [10–15]. The motivation is to capture access patterns that cannot be captured by stream prefetchers, namely those involving

indirections through memory as seen in sparse tensor algebra and graphs [13–15], and more generally applications with pointer chasing [10, 11]. The target application influences the data access pattern that the prefetcher tries to identify and prefetch for. For example, a common access pattern in sparse tensor algebra and graph computations is $\mathsf{An}[...\mathsf{A1}[\mathsf{A0}[\mathsf{i}]]...]$. Correspondingly, Yu et al. [13] (a.k.a. IMP) tries to detect access patterns given by $\mathsf{Y}[\mathsf{Z}[\mathsf{i}]]$ ("2-level IMP") and $\mathsf{X}[\mathsf{Y}[\mathsf{Z}[\mathsf{i}]]]$ ("3-level IMP"), for striding loop variable i, and prefetch data by assuming that $\mathsf{Y}[\mathsf{Z}[\mathsf{i}+\triangle]]$ and $\mathsf{X}[\mathsf{Y}[\mathsf{Z}[\mathsf{i}+\triangle]]]$ will be needed in the future. Ainsworth et al. [14] is similar, except with the pattern $\mathsf{W}[\mathsf{X}[\mathsf{Y}[\mathsf{Z}[\mathsf{i}]]]]$. Note that because such prefetchers typically need to cross page boundaries, they are typically located close to the core (to be able to access the TLB) and prefetch over virtual addresses.

3) **Security analysis:** Memory-centric optimizations have significant, novel security implications relative to the other optimizations we study because they leak data regardless of how it is computed. This falls outside the model for writing constant-time programs, and indiscriminately puts either all architectural registers or data memory at risk. Further, while we are unaware of current efforts towards implementing register-file compression, data memory-dependent prefetchers have been patented by industry [18].

4) **Building universal read gadgets, and analysis of data memory-dependent prefetcher variants:** Beyond simply leaking data memory, we showed in Section I how data-memory dependent prefetchers can additionally be used to create *universal read gadgets* (URGs). (We give more details on such attacks in Section V-B.)

Using MLD terminology, a URG is an optimization that takes data memory (data_memory) and attacker-controlled state c as input—and produces a different observable outcome as a function of $\mathsf{data\_memory}[f(\mathsf{c})]$, where f is an optimization-dependent attacker-known function. The more distinct observable outcomes for a given $f(\mathsf{c})$, the more the attacker can reduce its uncertainty about the value stored in data memory at that address. The larger the range of $f(\mathsf{c})$, the more values in memory the attacker can learn.

We can use the above framework to determine when optimizations, e.g., prefetchers, can form URGs and under what circumstances. For example, Figure 3, Example 9 shows the MLD for the 3-level IMP in [13]. The 2-level variant is the same except without the cache access to array X, offset y. Which of these forms a URG and why? Consider the sandbox setting (Section II-3). The attacker-controlled sandbox is the address range $[\mathsf{a}, \mathsf{b})$; private victim memory is outside of this range. Typically, s along with the base of Z, Y and X would be public. Thus the MLD indicates that the prefetcher reveals values s, z and y—offset by public values—through a cache-based transmitter. Since s is public, we focus on what the attacker can learn by observing z and y. In the IMP, $\mathsf{z} = \mathsf{Z}[\mathsf{i}+\triangle]$, i.e., is "nearby" data the attacker accessed recently. Then z can only represent values in the address range $[\mathsf{a}, \mathsf{b}+\triangle)$. On the other hand, it is easy to see that y may represent values in the address range $[\&\mathsf{Y}[0], \mathsf{MAX\_MEM\_ADDR})$, which is all of victim memory in the worst case, because $\mathsf{a} \leq \mathsf{z} < \mathsf{b}$ may hold and the attacker controls all data within that range. This tells us two things. First, that the 3-level IMP creates a URG. Second, that the 2-level IMP does not create a URG, beyond leaking victim data in the address range $[\mathsf{b}, \mathsf{b}+\triangle)$.

V. Evaluation: Proof-of-Concept Demonstrations

We now demonstrate proof-of-concept attacks on several of the optimizations we study, namely silent stores (Section IV-C) and data memory-dependent prefetching (Sections I, IV-D). While we cannot evaluate all optimizations we study, we chose these as exemplars based on the novelty of their security implications and their likelihood to be implemented in current/future processors.

A. Silent stores

We choose to evaluate silent stores because it is not obvious that their impact on the pipeline will lead to a measurable difference in the victim program's execution. Out-of-order pipelines are very good at preventing stalls due to stores being performed to memory. There are also nuances to a silent store design's implementation which can absorb data-dependent timing differences.

Despite these challenges, we demonstrate how a single dynamic instance of a secret key-dependent silent store can induce an end-to-end timing difference on a real world constant-time encryption function [104]. The key ingredient is a novel *amplification gadget*, i.e., a novel preconditioning for silent stores, which creates a large (> 100 cycles) timing difference depending on whether an attacker-chosen store is silent or not. We describe how this gadget can be leveraged to reveal the value of a critical store in an encryption operation and how this information can be used to reconstruct the victim's key.

1) **Silent stores implementation:** Lacking a real-world silent stores implementation, we implement silent stores and evaluate our attack in Gem5 [105]. Our implementation follows Lipasti et. al.'s proposal in [86]. As in [86], we assume a release-consistency memory model. Our amplification gadget also requires the store queue (SQ) to perform stores to memory (i.e., dequeue from the SQ) in program order. This implementation appears in processors today, e.g., the RISC-V BOOM processor [106]. Note, stores being performed do not dequeue from the SQ until the cache line they are writing to is present in the (first-level) cache.

We implement the read-port stealing scheme from [86]. Important cases are shown in Figure 4. The idea is to issue a load (called the Silent-Store-Load or *SS-Load*), as soon as the store address resolves and there is a free load port, that reads the contents of memory at the store address. If the SS-Load returns before the store is performed to memory, the store is marked silent iff the data returned by the SS-Load equals the store data. When a store marked silent is ready to be performed, it is silently dequeued without interacting with memory (Figure 4, Case A). If multiple consecutive stores are silent, they can all dequeue in the same cycle. Stores not marked silent are performed as usual (Case B). Note, this

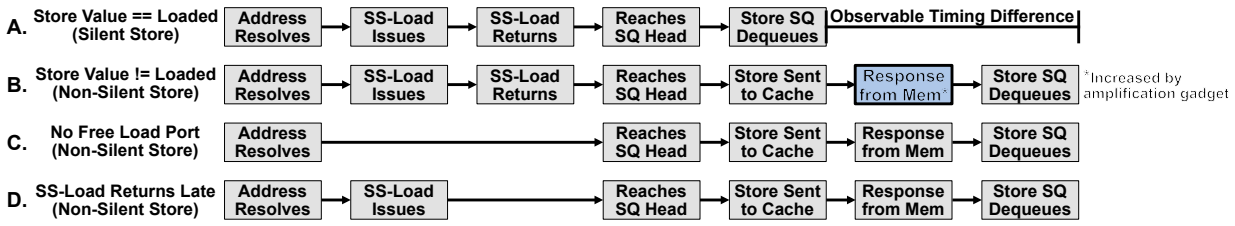

Fig. 4: The different possible sequences of actions taken by a store given the read-port stealing scheme from [86].

```
// Preconditions:
// - A, S have resolved
// - line(A) not present in cache
// - line(S) present in cache
// - set(S) != set(A), set(S) == set(A')
load A' <- (A)  // delay sub-gadget
load _  <- (A') // flush sub-gadget
store D -> (S)  // target store
```

Fig. 5: A single-threaded (i.e., inline with the victim program) amplification gadget for a release-consistency memory model and a direct-mapped cache. `line(X)` refers to the cache line associated with address `X`. `set(X)` refers to the cache set occupied by `line(X)`.

design does not change the time at which the store reaches the head of the SQ and only changes the store's behavior when it is dequeued.

Finally, if the SS-Load cannot issue because there is no free load port (Figure 4, Case C) or the SS-Load issues but arrives after the store is performed (Case D), the store is not marked silent. Case C is operationally equivalent to an architecture that does not implement silent stores.

2) **Amplification gadget:** While silent stores do change hardware resource usage depending on whether in-flight store data equals the contents of memory, it is non-trivial to convert these effects into measurable timing differences that contribute to end-to-end attacks. Making matters worse, an attack may hinge on the attacker observing a timing difference stemming from a *single dynamic store instruction*. Yet, the net effect of silent stores on a single dynamic store is likely completely hidden by out-of-order execution.

We address these challenges by describing an *amplification gadget* whose goal is to create a maximal timing difference based on whether a *single* specified dynamic store instruction is silent. The key idea is to maximize the time it takes the store in question (which we call the *target store*) to dequeue from the SQ in the case where it is not silent. If this dequeue time is sufficiently large, the SQ will fill and stall the pipeline. (Since stores are performed in program order (Section V-A1), long-to-dequeue stores will head-of-line block the SQ.) Thus, if all goes to plan, the amplification gadget stalls the pipeline iff the target store is not silent.

To maximally delay store dequeue time, the gadget arranges for the target store's cache line (called the *target line*) to not be present in the cache when the target store reaches the head of the SQ. Since dequeue only happens when the target line is filled in the cache (Section V-A1), this creates a delay proportional to the cache miss latency.

Creating the above gadget is challenging because the target store can only be a silent store candidate if the SS-Load returns before the target store is performed. That is, we must be in Cases A or B (not C or D) from Figure 4. Yet, this implies that the target line *is* present in the cache which is the opposite of what we wanted in the previous paragraph. To work around this issue, the amplification gadget must evict the target line *after* the SS-Load completes but *before* the target store is performed.

The amplification gadget's implementation depends on the threat model. We show one possible implementation where the gadget is a part of the victim program itself in Figure 5. The gadget is made up of the target store, as well as two (or more) other instructions that we call the delay sub-gadget and the flush sub-gadget (delay and flush gadgets for short). The delay gadget can be any instruction(s) that takes a long time to execute. The flush gadget can be any instruction(s) that depends on the delay gadget (or otherwise executes after the SS-Load for the target store returns; see below) and whose execution removes the target line from the cache.

Figure 5 gives an example implementation where both delay and flush gadgets are loads and flushing is implemented via cache set contention. Conceptually, the delay gadget will ensure that the SS-Load completes before the target store is performed and the flush gadget will ensure that the target line is evicted between SS-Load completion and when the target store is performed. In more detail, suppose these instructions execute. The delay gadget is performed and misses in the cache. Concurrently, the SS-Load (not shown in the pseudo-code) for the target store issues and returns first. Thus, the silent store candidacy check occurs before the target store is performed. Next, after the SS-Load completes, the delay gadget returns which enables the flush gadget to execute. Thus, the target line is removed from cache after the SS-Load completes but before the target store is performed, as desired.

Other implementations of the gadget are possible. For example, by having another co-resident thread emulate the flush gadget.

3) **Attacking Bitslice AES128 Encrypt:** Using the amplification gadget, we now demonstrate an end-to-end proof-of-concept attack on a "constant time" Bitslice AES128 encrypt

implementation [104] (abbreviated *BSAES*).

We design our proof-of-concept around a scenario in the cloud threat model (Section II-3). Our victim consists of a server with a worker thread for encryption calls. Both the adversary and victim trigger encryption calls with known plaintexts. Under this model, the encryption worker allocates various temporary variables on the stack. These variables are not cleared after use, as subsequent calls are guaranteed to overwrite them. This is the as-provided behavior of the victim program.

The goal of our attack is for the attacker to compute the victim's secret key by changing its own plaintext to induce a silent store at a specific location, conditioned on the data left behind in memory by the victim's prior encryption operation. We assume that the attacker has access to its own secret key, and that the victim is repeatedly encrypting the same public data (e.g., a packet header).

BSAES achieves constant-time byte substitution by performing a series of exclusive-or operations on the current AES state. Because this stage requires many more intermediate values than there are logical registers in x86, these values are spilled onto the stack. We identified eight locations storing intermediate values that can be used to reconstruct the AES state after byte substitution.

After the last byte substitution stage, BSAES performs an exclusive-or between the AES state and the last sixteen bytes of the expanded key to compute the final encrypted data. The key expansion algorithm is invertible, so knowing those sixteen bytes allows the attacker to reconstruct the entire original key [107].

The attack can be carried out as follows. First, the victim's data is encrypted, leaving behind AES state from its last byte substitution on the stack. Then, the attacker's data is encrypted and the attacker attempts to measure if any of the eight writes to intermediate values trigger silent stores. Because the attacker knows its own key and can modify its plaintext, it can try each possible value for each of the intermediate values over multiple experiments. Since these intermediate values are 16 bits, an attacker may need to try up to 65,536 possibilities for each value it wants to reconstruct, for a total of, at most, 524,288 attempts to compute the victim's key.

We use the amplification gadget to increase the timing difference caused by a single dynamic silent store on the BSAES code. For simplicity, we manually added the delay gadget and flush gadget (Section V-A2) to the BSAES code before the target store (one of the eight stores writing the AES state to the stack). In a real attack, the victim program would need a latent amplification gadget or the attacker would need to implement the gadget using co-resident threads (Section V-A2). Results are shown in Figure 6. Incorrect vs. correct corresponds to whether the attacker overwrites the AES state byte with a different vs. same value as was stored there before. This experiment assumes an architecture with a 5-entry SQ and a 4-way set associative cache. The takeaway is that depending on whether a single dynamic store is silent creates a large, easily distinguishable, timing difference (> 100 cycles).

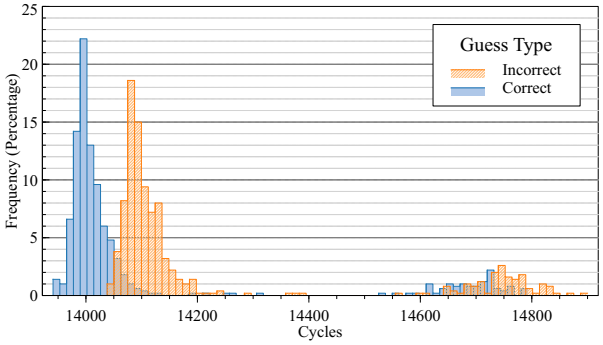

Fig. 6: Histogram of runtimes for BSAES when the amplification gadget is applied to one of the eight stores that overwrites AES state. That is, timing differences reflect whether a single dynamic store instruction was silent or not.

B. Data memory-dependent prefetching

The goal of this section is to provide details on the attack example we gave in Section I, which showed the 3-level indirect-memory prefetcher (IMP) [13, 18] leaking all victim memory, i.e., forming a universal read gadget. While we analyze the 3-level IMP here, we expect a similar attack to go through using any data-dependent memory prefetcher that performs at least two-level indirections (Section IV-D).

As in the example, we assume the attacker's goal is to learn kernel memory. Similar to the original Spectre PoC [21], the attacker runs a malicious program inside the eBPF sandbox and will use the IMP to break out of the sandbox (Section II-3). As discussed previously, eBPF performs static analysis on sandbox code, e.g., memory-safety checks, to ensure that it cannot break out of the sandbox. It then JITs the sandbox code to the target machine ISA to improve performance. Since we do not have access to real hardware running the IMP, we instead check our attacks' assumptions against the IMP design and eBPF. Specifically, (1) Can the attacker write a program that is a) likely to trigger IMP and b) bypasses eBPF's software analysis? (2) Does the attacker program cause the 3-level IMP to make out-of-bounds accesses?

1) **Bypassing eBPF:** We experimented with eBPF and found that it allows the program shown in Figure 7 to be run in the kernel. Figure 7a shows source code. In this example, the attacker can place the address of the value it wants to learn outside of the sandbox, i.e., the target (Section I), in $Z[N-1]$, or immediately out-of-bounds of Z. We found that eBPF complains unless one adds explicit NULL dereference checks, i.e., the "if (!v)" incantations, after each array lookup. These are bounds checks in disguise because an out-of-bounds .lookup() into a BPF_ARRAY returns NULL. We note that although eBPF requires programmers to use the BPF_ARRAY wrapper to instantiate arrays, Figure 7b shows how these get JITed into x86 assembly as if they are normal arrays. Specifically, we see no additional memory accesses made in between reading Z[i] and Y[Z[i]] into the register file, which is relevant below.

```
1 BPF_ARRAY(Z, int, N);
2 BPF_ARRAY(Y, int, N);
3 BPF_ARRAY(X, int, N);
4 int attacker () {
5   int j = 0; int *v;
6   for (j=0;j<N-1;j++) {
7     int i = j;
8     v = Z.lookup(&i);
9     if (!v) return 0;
10    v = Y.lookup(v);
11    if (!v) return 0;
12    v = X.lookup(v);
13    if (!v) return 0;
14    if (!*v) return 0; }
15  return 0; }
```

(a) Attacker eBPF source.

```
1 mov    0x0(%rsi),%eax # eax = Z[i]
2    # bounds check Z[i] < len(Y):
3 cmp    $0x40,%rax
4 jae    0x000000000000007f
5 shl    $0x3,%rax
6 add    %rdi,%rax # rax = &Y[Z[i]]
7 jmp    0x0000000000000081
8 xor    %eax,%eax
9 cmp    $0x0,%rax
10 je    0x00000000000000db
11 movabs $0x...,%rdi # rdi = X object
12 mov   %rax,%rsi
13 add   $0xd0,%rdi # rdi = &X array
14    # eax = Y[Z[i]]:
15 mov   0x0(%rsi),%eax
```

(b) Attacker eBPF JITed assembly snippet.

Fig. 7: Attacker program to break out of the eBPF sandbox using the 3-level indirect-memory prefetcher.

2) **Triggering out-of-bounds accesses using the 3-level IMP:** The program in Figure 7 is "tailor made" to trigger IMP. On the other hand, why would IMP make out-of-bounds accesses? In short, being a hardware prefetcher, IMP has no knowledge of array bounds. The IMP design [13] identifies indirect access patterns by monitoring data returned to the core (e.g., Z[i]s; Figure 7b, Line 1) and addresses for indirections that the core subsequently makes (e.g., Y[Z[i]]s; Figure 7b, Line 15). In the event that its target access pattern X[Y[Z[i]]] is under way, it will be able to look at this information to solve for the base of each array, i.e., &Y[0] and &X[0]. At this point, the IMP begins prefetching and *assumes* subsequent Z[i]s will remain in bounds—an assumption the attacker violates.

3) **Prefetch buffers:** Given the above discussion, we can conclude the attack will trigger the 3-level IMP to break the sandbox and transmit out-of-bounds data through a cache-based covert channel. One remaining subtlety is that some prefetchers use prefetch buffers, which prevent cache fills unless the prefetched data is actually read by the attacker program (which it will not be, per eBPF's checks). Such a "defense" might aggravate, but not mitigate our attack. For example, to our knowledge prefetch buffers are not applied to every cache level (e.g., the LLC), and the receiver (attacker) can simply monitor any un-buffered level.

VI. Discussion

A. Possible defense strategies and open problems

The optimization classes we study have privacy implications for data sent to a variety of instruction operands and for data at rest (Section III). Thus it seems imperative to invest in holistic, as opposed to point, defenses. We review potential approaches below, and discuss open challenges. *In particular, is it possible to get security back while maintaining all (or most) of the performance benefit brought by each optimization?*

1) **Open challenges in space-time sandboxes: can we efficiently protect every bit?:** There are a number of holistic defenses that attempt to block all possible microarchitectural attacks [45, 73, 108–110] through either spatial/temporal isolation [108–110] and/or information flow principles [45, 73]. For example: By isolating a victim spatially while it runs, one can block a receiver from measuring data-dependent fine-grain hardware resource usage. To hide termination time, one can pad execution time to the worst case [45, 73, 108, 109], or simply accept that seemingly-smaller amount of leakage [110].

While the above approaches hold significant promise, we argue that termination channel leakage creates a significant roadblock towards getting both security and performance in the context of the optimizations we study. That is, incurring worst-case execution time likely negates the optimizations' benefits. On the other hand, leaking even a single bit through timing rules out protecting many critical applications due to active replay attacks (Sections II-2, IV-D4)—i.e., where the attacker runs many experiments, changing data under its control for each experiment, to slowly leak the entire secret [1, 4].

2) **Retrofitting constant-time programming:** Current practices for constant-time programming [42–46] break under the optimizations we studied (Section III). Yet, it may be possible to retrofit these techniques to get back security. For example, to mitigate leakage from significance compression (Section IV-B), software instrumentation can OR a 1 into the most-significant bit position of each word (assuming this can be done while preserving functionality). To mitigate leakage through data memory-centric optimizations such as some variants of silent store attacks (Section IV-C) and data memory-dependent prefetching (Section IV-D), one can encrypt all data that is spilled from the register file/written to data memory. The advantage of these approaches is that they can be done in software. The disadvantages are complexity, brittleness and performance overhead.

It is prudent to start thinking about this retrofit now. For example, it is unclear (to us) how to retrofit constant-time programming to handle certain optimizations (e.g., value prediction or register-file compression). Further, there are likely ample opportunities for higher-performance mitigations given closer study. For example, while a naive method to block leakage through silent stores is to use encryption (see above), it may be sufficient to clear data memory in a targeted fashion.

Finally, we note that some ISAs are adding support for constant-time programming, e.g., ARM DIT [111]. This has promise to significantly simplify constant-time reasoning, but must be augmented to cover all optimizations we study. For example, ARM DIT as currently described does not cover memory-centric optimizations (Section IV-D).[6]

3) **Architecting security-conscious microarchitecture:** Finally, we would like to set a research agenda for proactively architecting security-conscious microarchitecture. The important observation is that, in some cases, slight tweaks to microarchitecture can render it more secure without compromising (much) performance. For example, dynamic instruction reuse [74] proposes several variants: S_v, S_n and S_{n+d}. The

[6]Interestingly, current ARM DIT does explicitly mention that timing should be independent of load/store data [111], indicating that ARM could potentially support protection against load-value prediction and silent stores (Section III).

S_v scheme uses instruction operand values as memoization table keys and thus can leak operand values (Section IV-C), whereas the latter two schemes use *operand register IDs* and thus only leak information about what instruction is executing (which is public information in constant-time programming, and only leaks control-flow related information more generally [56, 59, 69]). The paper [74] reports that S_v achieves the best performance, but that substantial performance gain is still possible with other variants. The takeaway is that we know how to, in some instances, architect still efficient and more secure microarchitecture. Going forward, the question is to what extent do similar principles generalize to other microarchitectural optimizations?

B. Additional optimizations with novel security implications

While we tried to be comprehensive, our study may be incomplete and we implore the community to continue thinking about and searching for microarchitecture with novel security implications. For example, many implemented optimizations may be unpublished, and outside of the optimizations we studied we did not search through industry patents. Further, there may be optimizations that create novel privacy implications only under subtle circumstances, that our analysis missed.

For example, dynamic optimization of program traces & hot regions [112, 113] and continuous optimization on instructions & micro-ops [29, 114] use runtime information to optimize program hot regions and instructions in flight, respectively. While at first glance such optimizations might seem to cause privacy problems, they only create novel security implications in specific circumstances. For example, based on our analysis, many runtime optimizations such as constant folding (e.g., in [29]) do not leak privacy beyond program control flow, which readily leaks through known attacks (Section III). On the other hand, if one were to apply a strength reduction optimization based on the value of a specific operand, this would create a security issue because strength reduction manifests due to specific operand data beyond control flow and results in changes to arithmetic unit port contention (which has been exploited in past attacks [59, 69, 115]). These and other similar, subtle performance techniques deserve further study; case in point, limited forms of continuous optimization are implemented today (e.g., as micro-op fusion [116]).

C. Processor attack landscape going forward

Finally, we note that beyond "traditional" microarchitectural attacks, trends in Dennard scaling and Moore's law are forging a significantly broader threat surface related to how power/energy interact with microarchitecture. For example, research has demonstrated how undervolting vis. DVFS effects enable novel integrity attacks through microarchitecture [117, 118]. Recent work has also shown how co-located but isolated processes can communicate through power-related mechanisms such as processor turbo boost [71, 72, 119, 120].

We consider a security analysis, analogous to the one performed in this paper, on power/energy↔microarchitecture-related attacks to be important future work. Case in point, these attacks not only can be performed remotely (similar to traditional microarchitectural attacks), but largely evade current defenses (e.g., spatial isolation of microarchitectural structures; Section VI-A1). We applaud recent work to extend spatial isolation to include energy [121], but more work is needed to find less-invasive solutions that apply to general-purpose programs.

VII. Related Work

See previous sections for related work on microarchitectural attacks (Section II), their implications on program data privacy (Section III) and our study of novel vulnerabilities through previously-unstudied microarchitectural mechanisms (Section IV). We provide background on relevant defense literature in Section VI-A.

To our knowledge, the only other work that studies the privacy implications of emerging microarchitecture is Safecracker [4]. This work is complementary to our analysis. It studies compressed caches (e.g., [122]), which would be classified as Memory-centric optimizations (Section IV) that render data memory Unsafe (Section III), similar to the data memory-dependent prefetchers we study. Safecracker proposes a universal read gadget (URG) but for the cloud setting, whereas we propose one using data memory-dependent prefetchers for the sandbox setting (see Sections II-3 and IV-D4). Yet, our URG requires weak software assumptions: the attacker merely has to trigger the data memory-dependent prefetcher in a setting where it has control over the program (Section V-B). Safecracker requires strong software assumptions, namely for the victim to contain a latent buffer overflow vulnerability.

VIII. Conclusion

This paper performed a systematic study of the computer architecture literature through a security lens. We found a range of microarchitectural optimizations with novel security implications—ranging from ones as devastating as Spectre/Meltdown (but without relying on speculative execution) to ones that render constant-time programming ineffective, or in need of overhaul. They further implicate a number of Computer Architecture concepts, ranging from value locality to compressibility to prediction to prefetching. While we believe that many are not implemented in commercial machines today, some may indeed be; and others are seeing a resurgence in interest. In any case, given the slowing of Moore's law, it stands to reason that many could be implemented in the future and we should be ready.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback, and thank Dean Tullsen for many insightful conversations over the years. This work was partially funded by an Intel ISRA grant, NSF grants #1816226 and #1942888, and ISF grant #2005/17.

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, 2019, "Spectre attacks: Exploiting speculative execution," in *S&P'19*.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, 2018, "Meltdown: Reading kernel memory from user space," in *Security'18*.

[3] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, 2020, "Rambleed: Reading bits in memory without accessing them," in *S&P'20*.

[4] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking Secrets through Compressed Caches," in *ASPLOS'20*.

[5] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, 2019, "RIDL: Rogue in-flight data load," in *S&P'19*.

[6] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," Cryptology ePrint Archive, Tech. Rep., 2016.

[7] 2018, "Championship Value Prediction (CVP)."

[8] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, 2018, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *CCS'18*.

[9] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, 2016, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *CCS'16*.

[10] A. Roth, A. Moshovos, and G. S. Sohi, 1998, "Dependence based prefetching for linked data structures," *SIGOPS Oper. Syst. Rev.*

[11] R. Cooksey, S. Jourdan, and D. Grunwald, 2002, "A stateless, content-directed data prefetching mechanism," *SIGOPS Oper. Syst. Rev.*

[12] B. Falsafi and T. F. Wenisch, 2014, "A primer on hardware prefetching," *Synth. Lect. Comput. Archit.*

[13] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, 2015, "Imp: Indirect memory prefetcher," in *MICRO'15*.

[14] S. Ainsworth and T. M. Jones, 2016, "Graph prefetching using data structure knowledge," in *ICS'16*.

[15] S. Ainsworth and T. M. Jones, 2018, "An event-triggered programmable prefetcher for irregular workloads," in *ASPLOS'18*.

[16] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, 2018, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *MICRO'18*.

[17] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, 2019, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv (2019)*.

[18] X. Yu, C. J. Hughes, and N. R. Satish, "Hardware prefetcher for indirect access patterns," Patent US9 582 422B2, Feb. 28, 2017.

[19] 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*.

[20] "Linux ebpf." [Online]. Available: https://ebpf.io/

[21] J. Horn, 2018, "Reading privileged memory with a side-channel."

[22] D. A. Osvik, A. Shamir, and E. Tromer, 2006, "Cache attacks and countermeasures: The case of aes," in *CT-RSA'06*.

[23] Z. Gong, H. Ji, C. W. Fletcher, C. Hughes, S. Baghsorkhi, and J. Torrellas, "SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs," in *MICRO'20*.

[24] D. Brooks and M. Martonosi, 1999, "Dynamically exploiting narrow width operands to improve processor power and performance," in *HPCA'99*.

[25] K. M. Lepak and M. H. Lipasti, 2000, "On the value locality of store instructions," in *ISCA'00*.

[26] A. Roth and G. S. Sohi, 2000, "Register integration: A simple and efficient implementation of squash reuse," in *MICRO 33*.

[27] A. Perais and A. Seznec, 2016, "Eole: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance," *TOCS'16*.

[28] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero, 2004, "A content aware integer register file organization," in *ISCA'04*.

[29] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, 2005, "Continuous optimization," in *ISCA'05*.

[30] T. Downs, 2020, "Hardware Store Elimination," *Performance Matters*.

[31] A. Waterman and K. Asanović, "The risc-v instruction set manual volume i: Unprivileged isa," EECS Department, University of California, Berkeley, Tech. Rep. [Online]. Available: https://riscv.org/specifications/isa-spec-pdf/

[32] M. H. Lipasti and J. P. Shen, 1996, "Exceeding the dataflow limit via value prediction," in *MICRO'96*.

[33] Y. Wu, "Management of reuse invalidation buffer for computation reuse," Patent US7 383 543B2, Jun. 3, 2008.

[34] H. Wang, P. Wang, R. Kling, N. A. Chazin, and J. Shen, "Quantization and compression for computation reuse," Patent US7 069 545, Jun. 27, 2006.

[35] P. G. Emma, A. M. Hartstein, H. Jacobson, and W. R. Reohr, "Method and structure for asynchronous skip-ahead in synchronous pipelines," Patent US7 945 765, May 17, 2011.

[36] T. A. Chilimbi, O. Ruwase, and V. Seshadri, "Loop code processor optimizations," Patent US10 459 727, Oct. 29, 2019.

[37] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, 2015, "On subnormal floating point and abnormal timing," in *S&P'15*.

[38] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, 2009, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*.

[39] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, 2019, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO'19*.

[40] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci, 2019, "NDA: Preventing Speculative Execution Attacks at Their Source," in *MICRO'19*.

[41] J. Fustos, F. Farshchi, and H. Yun, 2019, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," *DAC'19*.

[42] D. J. Bernstein, 2006, "Curve25519: New diffie-hellman speed records," in *PKC'06*.

[43] D. J. Bernstein, 2005, "The poly1305-aes message-authentication code," in *FSE'05*.

[44] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, 2009, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P'09*.

[45] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, 2019, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS'19*.

[46] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, 2017, "Fact: A flexible, constant-time programming language," *SecDev'17*.

[47] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, 2018, "Dawg: A defense against cache timing attacks in speculative execution processors," in *MICRO'18*.

[48] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, 2017, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *FC'17*.

[49] Y. Yarom and K. Falkner, 2014, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *Security'14*.

[50] Y. Yarom, D. Genkin, and N. Heninger, 2016, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *IACR'16*.

[51] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, 2019, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE S&P*.

[52] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, 2018, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Security'18*.

[53] Y. Xu, W. Cui, and M. Peinado, 2015, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*.

[54] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, 2017, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *CCS'17*.

[55] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, 2016, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *Security'16*.

[56] O. Aciicmez, J.-P. Seifert, and C. K. Koc, 2006, "Predicting secret keys via branch prediction," *IACR'06*.

[57] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, 2018, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS'18*.

[58] A. Moghimi, T. Eisenbarth, and B. Sunar, 2017, "Memjam: A false dependency attack against constant-time crypto implementations," *CoRR'17*.

[59] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, 2018, "Port contention for fun and profit," in *IACR'18*.

[60] D. Evtyushkin and D. Ponomarev, 2016, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS '16*.

[61] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, 2020, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *MICRO'20*.

[62] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, 2019, "Netspectre: Read arbitrary memory over network," in *ESORICS'19*.

[63] P. C. Kocher, J. Jaffe, and B. Jun, 1999, "Differential power analysis," in *CRYPTO'99*.

[64] Intel, 2013, "Intel Software Guard Extensions Programming Reference."

[65] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[66] Y. Han, T. Alpcan, J. Chan, and C. Leckie, 2013, "Security games for virtual machine allocation in cloud computing," in *GameSec'13*.

[67] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, 2014, "Cross-tenant side-channel attacks in paas clouds," in *CCS'14*.

[68] J. Van Bulck, F. Piessens, and R. Strackx, 2017, "Sgx-step: A practical attack framework for precise enclave execution control," in *SysTEX'17*.

[69] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, 2019, "Microscope: Enabling microarchitectural replay attacks," in *ISCA'19*.

[70] A. P. Sayakkara, N. Le-Khac, and M. Scanlon, 2019, "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics," *CoRR'19*.

[71] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu, 2019, "Powert channels: A novel class of covert communicationexploiting power management vulnerabilities," in *HPCA'19*.

[72] M. Kalmbach, M. Gottschlag, T. Schmidt, and F. Bellosa, 2020, "Turbocc: A practical frequency-based covert channel with intel turbo boost," *arXiv (2020)*.

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

[74] A. Sodani and G. S. Sohi, 1997, "Dynamic instruction reuse," in *ISCA'97*.

[75] S. Mittal, 2017, "A survey of value prediction techniques for leveraging value locality," *CCPE'17*.

[76] S. Balakrishnan and G. Sohi, 2003, "Exploiting value locality in physical register files," in *MICRO'03*.

[77] S. E. Richardson, 1993, "Exploiting trivial and redundant computation," in *ARITH'93*.

[78] J. J. Yi and D. J. Lilja, 2002, "Improving processor performance by simplifying and bypassing trivial computations," in *ICCD'02*.

[79] E. Atoofian and A. Baniasadi, 2005, "Improving energy-efficiency by bypassing trivial computations," in *IPDPS'05*.

[80] M. M. Islam and P. Stenstrom, 2006, "Reduction of energy consumption in processors by early detection and bypassing of trivial operations," in *SAMOS'06*.

[81] S. Kim, 2007, "Reducing alu and register file energy by dynamic zero detection," in *IPCCC'07*.

[82] M. M. Islam and P. Stenstrom, 2007, "Energy and performance tradeoffs between instruction reuse and trivial computations for embedded applications," in *SIES'07*.

[83] R. Canal, A. González, and J. E. Smith, 2000, "Very low power pipelines using significance compression," in *MICRO'00*.

[84] V. N. Ekanayake, C. Kelly, and R. Manohar, 2005, "Bitsnap: Dynamic significance compression for a low-energy sensor network asynchronous processor," in *ASYNC'05*.

[85] S. Kumar, P. Pujara, and A. Aggarwal, 2004, "Bit-sliced datapath for energy-efficient high performance microprocessors," in *PACS'05*.

[86] K. Lepak and M. Lipasti, 2000, "Silent stores for free," in *MICRO'00*.

[87] I. Kim and M. H. Lipasti, 2002, "Implementing optimizations at decode time," in *ISCA'02*.

[88] S. P. Harbison, 1982, "An architectural alternative to optimizing compilers," in *ASPLOS I*.

[89] S. E. Richardson, *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*, 1992.

[90] S. Oberman and M. J. Flynn, "On division and reciprocal caches," Tech. Rep., 1995.

[91] A. Sodani and G. Sohi, 1998, "Understanding the differences between value prediction and instruction reuse," in *MICRO'98*.

[92] C. Molina, A. González, and J. Tubella, 1999, "Dynamic removal of redundant computations," in *ICS'99*.

[93] A. Gonzalez, J. Tubella, and C. Molina, 1999, "Trace-level reuse," in *ICPP'99*.

[94] K. R. Gandhi and N. R. Mahapatra, 2008, "Partitioned reuse cache for energy-efficient soft-error protection of functional units," in *SOCC'08*.

[95] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, 1996, "Value locality and load value prediction," *ASPLOS'96*.

[96] B. Calder and G. Reinman, 2000, "A comparative survey of load speculation architectures," *JILP'00*.

[97] A. Seznec, 2018, "Exploring value prediction with the eves predictor," in *CVP-Championship Value Prediction*.

[98] R. Sheikh, H. W. Cain, and R. Damodaran, 2017, "Load value prediction via path-based address prediction: avoiding mispredictions due to conflicting stores," in *MICRO'17*.

[99] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, 2019, "Efficient invisible speculative execution through selective delay and value prediction," in *ISCA'19*.

[100] C. Sakhuja, A. Subramanian, P. Joshi, A. Jain, and C. Lin, 2019, "Combining branch history and value history for improved value prediction," *CVP-Championship Value Prediction*.

[101] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, 1998, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *MICRO'98*.

[102] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang, 2004, "Dynamically reducing pressure on the physical register file through simple register sharing," in *ISPASS'04*.

[103] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, 2004, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *MICRO'04*.

[104] "Bitslice AES (Bitcoin)."

[105] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, 2011, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*.

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

[107] J. Bonneau, 01 2006, "Robust final-round cache-trace attacks against aes." *IACR Cryptology ePrint Archive*.

[108] Z. B. Aweke and T. M. Austin, 2017, "Ozone: Efficient execution with zero timing leakage for modern microarchitectures," *CoRR'17*.

[109] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, 2009, "Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference," in *MICRO'09*.

[110] T. Bourgeat, I. A. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, 2018, "MI6: secure enclaves in a speculative out-of-order processor," *CoRR'18*.

[111] "Arm architecture registers armv8, for armv8-a architecture profile."

[112] D. H. Friendly, S. J. Patel, and Y. N. Patt, 1998, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *MICRO'98*.

[113] S. J. Patel and S. S. Lumetta, 2001, "Replay: A hardware framework for dynamic optimization," *TC'01*.

[114] B. Slechta, D. Crowe, N. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. J. Patel, and S. S. Lumetta, 2003, "Dynamic optimization of micro-operations," in *HPCA'03*.

[115] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, 2019, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *CCS'19*.

[116] "Intel x86_64 and ia32 developers manual."

[117] A. Tang, S. Sethumadhavan, and S. Stolfo, Aug. 2017, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *Security'17*.

[118] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, 2020, "Plundervolt: Software-based fault injection attacks against intel sgx," in *S&P'20*.

[119] "Poc-gtfo 2019." [Online]. Available: https://www.alchemistowl.org/pocorgtfo/pocorgtfo19.pdf

[120] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, 2021, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," *IEEE S&P'21*.

[121] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. B. Taylor, and R. Kastner, 2018, "Hiding intermittent information leakage with architectural support for blinking," in *ISCA'18*.

[122] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, 2012, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT'12*.