2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors*

Vladimir Kiriansky†, Ilia Lebedev†, Saman Amarasinghe†, Srinivas Devadas†, Joel Emer‡
†*MIT CSAIL,* ‡*NVIDIA / MIT CSAIL*
{vlk, ilebedev, saman, devadas, emer}@csail.mit.edu

***Abstract*—Software side channel attacks have become a serious concern with the recent rash of attacks on speculative processor architectures. Most attacks that have been demonstrated exploit the cache tag state as their exfiltration channel. While many existing defense mechanisms that can be implemented solely in software have been proposed, these mechanisms appear to patch specific attacks, and can be circumvented. In this paper, we propose minimal modifications to hardware to defend against a broad class of attacks, including those based on speculation, with the goal of eliminating the entire attack surface associated with the cache state covert channel.**

We propose DAWG, Dynamically Allocated Way Guard, a generic mechanism for secure way partitioning of set associative structures including memory caches. DAWG endows a set associative structure with a notion of protection domains to provide strong isolation. When applied to a cache, unlike existing quality of service mechanisms such as Intel's Cache Allocation Technology (CAT), DAWG fully isolates hits, misses, and metadata updates across protection domains. We describe how DAWG can be implemented on a processor with minimal modifications to modern operating systems. We describe a non-interference property that is orthogonal to speculative execution and therefore argue that existing attacks such as Spectre Variant 1 and 2 will not work on a system equipped with DAWG. Finally, we evaluate the performance impact of DAWG on the cache subsystem.

I. Introduction

For decades, processors have been architected for performance or power-performance. While it was generally assumed by computer architects that performance and security are orthogonal concerns, there are a slew of examples, including the recent Google Project Zero attacks [22] (Spectre [31] and Meltdown [35]) and variants [30], that show that performance and security are not independent, and micro-architectural optimizations that preserve architectural correctness can affect the security of the system.

In security attacks, the objective of the *attacker* is to create some software that can steal some *secret* that another piece of code, the *victim*, should have exclusive access to. The access to the secret may be made directly, e.g., by reading the value of a memory location, or indirectly, e.g., inferred from the execution flow a program takes. In either case, this leakage of information is referred to as violating *isolation*, which is different from violating *integrity* (corrupting the results obtained through program execution).[1]

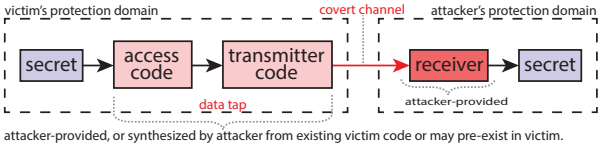

Fig. 1. Attack Schema: an adversary 1) accesses a victim's secret, 2) transmits it via a covert channel, and 3) receives it in their own protection domain.

In a well-designed system the attacker cannot architecturally observe this secret, as the secret should be confined to a *protection domain* that prevents other programs from observing it architecturally. However, vulnerabilities may exist when an attacker can observe side effects of execution via software means.

The mechanism by which such observations are made are referred to as *software side channels*. Such channels must be *modulated*, i.e., their state changed, as a function of activity in the victim's protection domain and the attacker must be able to detect those state changes. Currently, the most widely explored channel is based on the state of a shared cache. For example, if the attacker observes a hit on an address, the address must be cached already, meaning some party, maybe the victim, had recently accessed it, and it had not yet been displaced. Determining if an access is a hit can be accomplished by measuring the time it takes for a program to make specific references.

A *covert communication channel* transfers information between processes that should not be allowed to communicate by existing protection mechanisms. For example, when a side channel is used to convey a "secret" to an attacker, an attack would include code inside the victim's protection domain for accessing the secret and a *transmitter* for conveying the secret to the attacker. Together they form a *data tap* that will modulate the channel based on the secret. A *receiver* controlled by the attacker, and outside the victim's protection domain, will listen for a signal on the channel and decode it to determine the secret. This is pictorially illustrated in Fig. 1.

*Student and faculty authors listed in alphabetical order.

Funding for this research was partially provided by NSF grant CNS-1413920; DARPA contracts HR001118C0018, HR00111830007, and FA87501720126; Delta Electronics, DARPA & SPAWAR contract N66001-15-C-4066; DoE award DE-FOA0001059, and Toyota grant LP-C000765-SR. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon.

[1]Violating isolation and obtaining a secret may result in the attacker being able to violate integrity as well, since it may now have the capability to modify memory, but in this paper we focus on the initial attack that would violate isolation.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00083

A classic attack on RSA relied on such a scenario [9]. Specifically, existing RSA code followed a conditional execution sequence that was a function of the secret, and inadvertently transmitted private information by modifying instruction cache state in accord with that execution sequence. This resulted in a covert communication that let an observing adversary determine bits of the secret. In this case, the code that accessed the secret and the transmitter that conveyed the secret were pre-existing in the RSA code. Thus, an attacker that shared the icache needed only provide a receiver that could demodulate the secret conveyed over the cache tag state-based channel. Recent work has shown that a broad space of viable attacks exfiltrate information via shared caches.

A. Generalized Attack Schema

Recently, multiple security researchers (e.g., [22], [31], [35]) have found ways for an attacker to create a *new* data tap in the victim. Here, an attacker is able to *create* a data tap in the victim's domain and/or influences the data tap to access and transmit a chosen secret. Spectre and Meltdown have exploited the fact that code executing *speculatively* has full access to any secret.

While speculative execution is broadly defined, we focus on control flow speculation in this paper. Modern processors execute instructions *out of order*, allowing downstream instructions to execute prior to upstream instructions as long as dependencies are preserved. Most instructions on modern out-of-order processors are also *speculative*, i.e., they create checkpoints and execute along a predicted path while one or more prior conditional branches are pending resolution. A prediction resolved to be correct discards a checkpoint state, while an incorrect one forces the processor to roll back to the checkpoint and resume along the correct path. Incorrectly predicted instructions are executed, for a time, but do not modify architectural state. However, micro-architectural state such as cache tag state *is* modified as a result of (incorrect) speculative execution causing a channel to be modulated, which may allow secrets to leak.

By exploiting mis-speculated execution, an attacker can exercise code paths that are normally not reachable, circumventing software invariants. One example has the attacker speculatively executing data tap code that illegally accesses the secret and causes a transmission via micro-architectural side effects before an exception is raised [35]. Another example has the attacker coercing branch predictor state to encourage mis-speculation along an attacker-selected code path, which implements a data tap in the victim's domain. There are therefore three ways of creating the data tap:

1) Data tap pre-exists in victim's code, which we described in the RSA attack [9].
2) Attacker explicitly programs the data tap. Meltdown [35] is an example of this.
3) Attacker synthesizes a data tap out of existing code in the victim — exemplified by Spectre variants [22], [30], [31].

This framework can be applied for side channels other than the cache state, describing exfiltration via branch predictor logic or TLB state, for example. Given the intensified research interest in variants of this new attack class, we also imagine that there will be new ways that data taps can be constructed. We therefore wish to design a defense against a broad class of current and future attacks.

B. Our approach to defense

Defense mechanisms that can be implemented solely in software have been proposed (e.g., [11], [43]). Unfortunately, these mechanisms appear very attack specific: e.g., a compiler analysis [43] identifies some instances of code vulnerable to Spectre Variant 1; microcode updates or compiler and linker fixes reduce exposure to Spectre Variant 2 [11]. Instructions to turn off speculation in vulnerable regions have been introduced (e.g., [2]) for future compilers to use. In this paper, we target minimal modifications to hardware that defend against a broad class of side channel attacks, including those based on speculation, with the goal of eliminating the entire attack surface associated with exfiltration via changing cache state.

To prevent exfiltration, we require strong isolation between protection domains, which prevents any transmitter/receiver pair from sharing the same channel. Cache partitioning is an appealing mechanism to achieve isolation. Unfortunately, set (e.g., page coloring [29], [50]) and way (e.g., Intel's Cache Allocation Technology (CAT) [21], [23]) partitioning mechanisms available in today's processors are either low-performing or do not provide isolation.

We propose DAWG, *Dynamically Allocated Way Guard*, a generic mechanism for secure way partitioning of set associative structures including caches. DAWG endows a set associative structure with a notion of *protection domains* to provide strong isolation. Unlike existing mechanisms such as CAT, DAWG disallows hits across protection domains. This affects hit paths and cache coherence [42], and DAWG handles these issues with minimal modification to modern operating systems, while reducing the attack surface of operating systems to a small set of annotated sections where data moves across protection domains, or where domains are resized/reallocated. Only in these handful of routines, DAWG protection is relaxed, and other defensive mechanisms such as speculation fences are applied as needed. We evaluate the performance implications of DAWG using a combination of architectural simulation and real hardware and compare to conventional and quality-of-service partitioned caches. We conclude that DAWG provides strong isolation with reasonable performance overhead.

C. Contributions and organization

The contributions of our paper are:

1) We motivate strong isolation of replacement metadata by demonstrating that the replacement policy can leak information (cf. Section II-B2) in a way-partitioned cache.
2) We design a cache way partitioning scheme, DAWG, with strong isolation properties that blocks old and new attacks based on the cache state exfiltration channel (cf. Section III). DAWG does not require invasive changes to modern

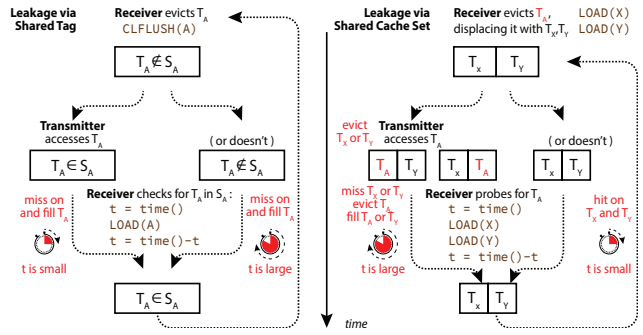

Fig. 2. Leakage via a shared cache set, implemented via a shared tag T_A directly, or indirectly via $T_X, T_Y \cong T_A$.

operating systems, and preserves the semantics of copy-on-write resource management.

3) We analyze the security of DAWG and argue its security against recent attacks that exploit speculative execution and cache-based channels (cf. Section V-A).
4) We illustrate the limitations of cache partitioning for isolation by discussing a hypothetical leak framed by our attack schema (cf. Fig. 1) that circumvents a partitioned cache. For completeness, we briefly describe a defense against this type of attack (cf. Section V-C).
5) We evaluate the performance impact of DAWG in comparison to CAT [21] and non-partitioned caches with a variety of workloads, detailing the overhead of DAWG's protection domains, which limit data sharing in the system (cf. Section VI).

The paper is organized as follows. We provide background and discuss related work in Section II. The hardware modifications implied by DAWG are presented in Section III, and software support is detailed in Section IV. Security analysis and evaluation are the subjects of Section V and Section VI, respectively. Section VII concludes.

II. Background and Related Work

We focus on thwarting attacks by disrupting the channel between the victim's domain and the attacker for attacks that use cache state-based channels. We state our threat model in Section II-A, describe relevant attacks in Section II-B, and existing defenses in Section II-C.

A. Threat model

Our focus is on blocking attacks that utilize the cache state exfiltration channel. We do not claim to disrupt other channels, such as L3 cache slice contention, L2 cache bank contention, network-on-chip or DRAM bandwidth contention, branch data structures, TLBs or shared functional units in a physical core. In the case of the branch data structures, TLBs, or any other set associative structure, however, we believe that a DAWG-like technique can be used to block the channel associated with the state of those structures. We assume an unprivileged attacker. The victim's domain can be privileged (kernel) code or an unprivileged process.

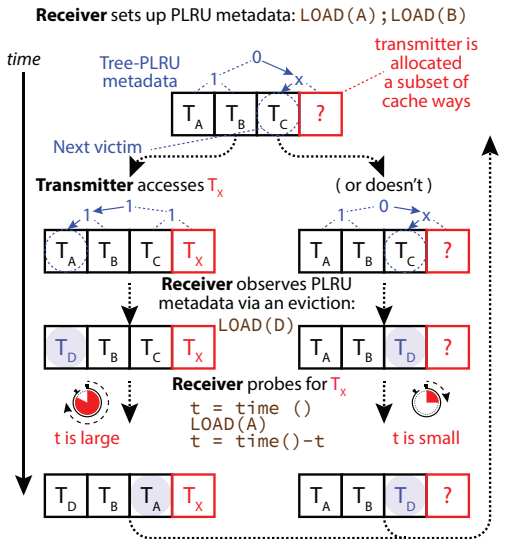

Fig. 3. Covert channel via shared replacement metadata, exemplified by a 4-way set-associative cache with a Tree-PLRU policy, cache allocation boundary. Tags $T_A \cong T_B \cong T_C \cong T_D \cong T_X$.

B. Attacks

The most common channel modulation strategy corresponds to the attacker presetting the cache tag state to a particular value, and then after the victim runs, observing a difference in the cache tag state to learn something about the victim process. A less common yet viable strategy corresponds to observing changes in coherence [59] or replacement metadata.

1) Cache tag state based attacks: Attacks using cache tag state-based channels are known to retrieve cryptographic keys from a growing body of cryptographic implementations: AES [7], [40], RSA [9], Diffie-Hellman [32], and elliptic-curve cryptography [8], to name a few. Such attacks can be mounted by unprivileged software sharing a computer with the victim software [3]. While early attacks required access to the victim's CPU core, more recent sophisticated channel modulation schemes such as flush+reload [60] and variants of prime+probe [38] target the last-level cache (LLC), which is shared by all cores in a socket. The evict+reload variant of flush+reload uses cache contention rather than flushing [38]. An attack in JavaScript that used a cache state-based channel was demonstrated [39] to automatically exfiltrate private information upon a web page visit.

These attacks use channels at various levels of the memory cache hierarchy and exploit cache lines shared between an attacker's program and the victim process. Regardless of the specific mechanism for inspecting shared tags, the underlying concepts are the same: two entities separated by a trust boundary share a channel based on shared computer system resources, specifically sets in the memory hierarchy. Thus, the entities can communicate (transmitting unwittingly, in the case of an attack) on that cross-trust boundary channel by modulating the presence of a cache tag in a set. The receiver can detect the transmitter's fills of tag T_A either directly, by

observing whether it had fetched a shared line, or indirectly, by observing conflict misses on the receiver's own data caused by the transmitter's accesses, as shown in Fig. 2.

2) A cache metadata-based channel: Even without shared cache lines (as is the case in a way-partitioned cache), the replacement metadata associated with each set may be used as a channel. Most replacement policies employ a replacement state bit vector that encodes access history to the cache set in order to predict the ways least costly to evict in case of a miss. If the cache does not explicitly partition the replacement state metadata across protection domains, some policies may violate isolation in the cache by allowing one protection domain's accesses to affect victim selection in another partition. Fig. 3 exemplifies this with Tree-PLRU replacement (Section III-J1): a metadata update after an access to a small partition overwrites metadata bits used to select the victim in a larger partition. A securely way-partitioned cache must ensure that replacement metadata does not allow information flow across the cache partition(s).

This means defenses against cache channel-based attacks have to take into account the cache replacement policy and potentially modify the policy to disrupt the channel and hence ensure isolation.

C. Defenses

Broadly speaking, there are five classes of defenses, with each class corresponding to blocking one of the steps of the attack described in Fig. 1.

1) *Prevent access to the secret*. For example, KAISER [13], which removes virtual address mappings of kernel memory when executing in user mode, is effective against Meltdown [35].
2) *Make it difficult to construct the data tap*. For example, randomizing virtual addresses of code, flushing the Branch Table Buffer (BTB) when entering victim's domain [46].
3) *Make it difficult to launch the data tap*. For example, not speculatively executing through permission checks, keeping predictor state partitioned between domains, and preventing user arguments from influencing code with access to secrets. The Retpoline [53] defense against Spectre Variant 2 [11] makes it hard to launch (or construct) a data tap via an indirect branch.
4) *Reduce the bandwidth of side channels*. For example, removing the APIs for high resolution timestamps in JavaScript, as well as support for shared memory buffers to prevent attackers from creating timers.
5) *Close the side channels*. Prevent the attacker and victim from having access to the same channel. For example, partitioning of cache state or predictor state.

The latter is the strategy of choice in our paper, and we consider three subclasses of prior approaches:

1) Set partitioning via page coloring: Set partitioning, i.e., not allowing occupancy of any cache set by data from different protection domains, can disrupt cache state-based channels. It has the advantage of working with existing hardware when allocating groups of sets at page granularity [34], [61] via page coloring [29], [50]. Linux currently does not support page coloring, since most early OS coloring was driven by the needs of low-associativity data caches [51].

Set partitioning allows communication between protection domains without destroying cache coherence. The downsides are that it requires some privileged entity, or collaboration, to move large regions of data around in memory when allocating cache sets, as set partitioning via page coloring binds cache set allocation to physical address allocation. For example, in order to give a protection domain 1/8 of the cache space, the same 12.5% of the system's physical address space must be given to the process. In an ideal situation, the amount of allocated DRAM and the amount of allocated cache space should be decoupled.

Furthermore, cache coloring at page granularity is not straightforwardly compatible with large pages, drastically reducing the TLB reach, and therefore performance, of processes. On current processors, the index bits placement requires that small (4KB) pages are used, and coloring is not possible for large (2MB) pages. Large pages provide critical performance benefits for virtualization platforms used in the public cloud [44], and reverting to small pages would be deleterious.

2) Insecure way and fine-grain partitioning: Intel's Cache Allocation Technology (CAT) [21], [23] provides a mechanism to configure each logical process with a *class of service*, and allocates LLC cache ways to logical processes. The CAT manual explicitly states that a cache access will hit if the line is cached in *any* of the cache's ways — this allows attackers to observe accesses of the victim. CAT only guarantees that a domain fill will not cause evictions in another domain. To achieve CAT's properties, no critical path changes in the cache are required: CAT's behavior on a cache hit is identical to a generic cache. Victim selection (replacement policy), however, must be made aware of the CAT configuration in order to constrain ways on an eviction.

Via this quality of service (QoS) mechanism, CAT improves system performance because an inefficient, cache-hungry process can be reined in and made to only cause evictions in a subset of the LLC, instead of trashing the entire cache. The fact that the cache checks all ways for cache hits is also good for performance: shared data need not be duplicated, and overhead due to internal fragmentation of cache ways is reduced. The number of ways for each domain can also be dynamically adjusted. For example, DynaWay [16] uses CAT with online performance monitoring to adjust the ways per domain.

CAT-style partitioning is unfortunately insufficient for blocking all cache state-based channels: an attacker sharing a page with the victim may observe the victim's use of shared addresses (by measuring whether a load to a shared address results in a cache hit). Furthermore, even though domains can fill only in their own ways, an attacker is free to flush shared cache lines regardless where they are cached, allowing straightforward transmission to an attacker's receiver via flush&reload, or flush&flush [20]. CAT-style partitioning allows an attacker to spy on lines cached in ways allocated to the

victim, so long as the address of a transmitting line is mapped by the attacker. This is especially problematic when considering Spectre-style attacks, as the victim (OpenSSL, kernel, etc.) can be made to speculatively touch arbitrary addresses, including those in shared pages. In a more subtle channel, access patterns leak through metadata updates on hitting loads, as the replacement metadata is shared across protection domains.

Applying DAWG domain isolation to fine-grain QoS partitioning such as Vantage [47] would further improve scalability to high core counts. Securing Vantage, is similar to securing CAT: hits can be isolated, since each cache tag is associated with a partition ID; replacement metadata (timestamps or RRIP [26]) should be restricted to each partition; additionally Vantage misses allow interference, and demotion to the unmanaged 10% of the cache, which must be secured.

3) Reducing privacy leakage from caches: Since Spectre attacks are outside of the threat model anticipated by prior work, most prior defenses are ineffective. LLC defenses against cross-core attacks, such as SHARP [58] and RIC [28], do not stop same-core OS/VMM attacks. In addition, RIC's non-inclusive read-only caches do not stop speculative attacks from leaking through read-write cache lines in cache coherence attacks [52].

PLcache [33], [56] and the Random Fill Cache Architecture (RFill, [37]) were designed and analyzed in the context of a small region of sensitive data. RPcache [33], [56] trusts the OS to assign different hardware process IDs to mutually mistrusting entities, and its mechanism does not directly scale to large LLCs. The non-monopolizable cache [14] uses a well-principled partitioning scheme, but does not completely block all channels, and relies on the OS to assign hardware process IDs. CATalyst [36] trusts the Xen hypervisor to correctly tame Intel's Cache Allocation Technology into providing cache pinning, which can only secure software whose code and data fits into a fraction of the LLC, e.g., each virtual machine is given 8 "secure" pages. [49] similarly depends on CAT for the KVM (Kernel-based Virtual Machine) hypervisor. Using hardware transactional memory, Cloak [19] preloads secrets in cache within one transaction to prevent access pattern observation of secrets. Blocking channels used by speculative attacks, however, requires all addressable memory to be protected.

SecDCP [55] demonstrate dynamic allocation policies, assuming a secure partitioning mechanism is available; they provide only 'one-way protection' for a privileged enclave with no communication. DAWG offers the desired partitioning mechanism; we additionally enable two-way communication between OS and applications, and handle mutually untrusted peers at the same security level. We allow deduplication, shared libraries, and memory mapping, which in prior work must all be disabled.

III. Dynamically Allocated Way Guard (DAWG) Hardware

The objective of DAWG is to preclude the existence of any cache state-based channels between the attacker's and victim's domains. It accomplishes this by isolating the visibility of any state changes to a single protection domain, so any transmitter in the victim's domain cannot be connected to the same channel as any receiver in the attacker's domain. This prevents any communication or *leaks* of data from the victim to the attacker.

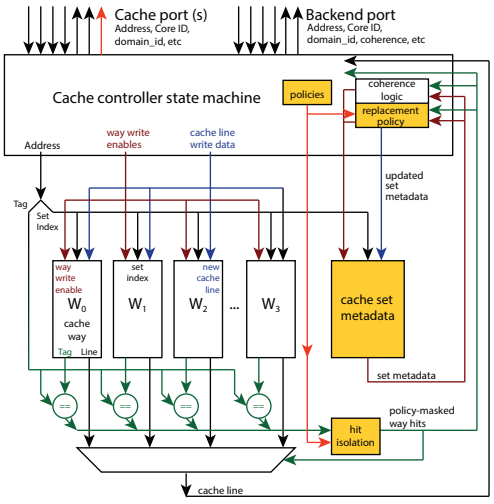

Fig. 4. A Set-Associative Cache structure with DAWG.

A. High-level design

Consider a conventional set-associative cache, a structure comprised of several *ways*, each of which is essentially a direct-mapped cache, as well as a controller mechanism. In order to implement Dynamically Allocated Way Guard (DAWG), we will allocate groups of ways to protection domains, restricting both cache hits and line replacements to the ways allocated to the protection domain from which the cache request was issued. On top of that, the *metadata* associated with the cache, e.g., replacement policy state, must also be allocated to protection domains in a well-defined way, and securely partitioned. These allocations will force strong isolation between the domains' interactions with one another via the cache structure.

DAWG's protection domains are *disjoint across ways and across metadata partitions*, except that protection domains may be nested to allow trusted privileged software access to all ways and metadata allocated to the protection domains in its purview.

Fig. 4 shows the hardware structure corresponding to a DAWG cache, with the additional hardware required by DAWG over a conventional set-associative cache shown highlighted. The additional hardware state for each core is 24 bits per hardware thread – one register with three 8-bit active domain selectors. Each cache additionally needs up to 256 bits to describe the allowed hit and fill ways for each active domain (e.g., 16× intervals for a typical current 16-way cache).

B. DAWG's isolation policies

DAWG's protection domains are a high-level property orchestrated by software, and implemented via a table of *policy* configurations, used by the cache to enforce DAWG's isolation;

these are stored at the DAWG cache in MSRs (model-specific registers). System software can write to these policy MSRs for each `domain_id` to configure the protection domains as enforced by the cache.

Each access to a conventional cache structure is accompanied with request metadata, such as a Core ID, as in Fig. 4. DAWG extends this metadata to reference a *policy* specifying the protection domain (`domain_id`) as context for the cache access. For a last-level memory cache the `domain_id` field is required to allow system software to propagate the domain on whose behalf the access occurs, much like a capability. The hardware needed to endow each cache access with appropriate `domain_id` is described in Section III-C.

Each policy consists of a pair of bit fields, all accessible via the DAWG cache's MSRs:

- A `policy_fillmap`: a bit vector masking fills and victim selection, as described in Sections III-D and III-E.
- A `policy_hitmap`: a bit vector masking way *hits* in the DAWG cache, as described in Section III-F.

Each DAWG cache stores a table of these policy configurations, managed by system software, and selected by the cache request metadata at each cache access. Specifically, this table maps global `domain_id` identifiers to that domain's policy configuration in a given DAWG cache. We discuss the software primitives to manage protection domains, i.e., to create, modify, and destroy way allocations for protection domains, and to associate processes with protection domains in Section IV-A1.

C. DAWG's modifications to processor cores

Each (logical) core must also correctly tag its memory accesses with the correct `domain_id`. To this end, we endow each hardware thread (logical core) with an MSR specifying the `domain_id` fields for each of the three types of accesses recognized by DAWG: instruction fetches via the instruction cache, read-only accesses (loads, flushes, etc), and modifying accesses (anything that can cause a cache line to enter the modified state, e.g., stores or atomic accesses). We will refer to these three types of accesses as *ifetches*, *loads*, and *stores*; (anachronistically, we name the respective domain selectors CS, DS, and ES). Normally, all three types of accesses are associated with the same protection domain, but this is not the case during OS handling of memory during communication across domains (for example when servicing a system call). The categorization of accesses is important to allow system software to implement message passing, and the indirection through domain selectors allows domain resizing, as described in Section IV.

The bit width of the `domain_id` identifier caps the number of protection domains that can be simultaneously scheduled to execute across the system. In practice, a single bit (differentiating kernel and user-mode accesses) is a useful minimum, and a reasonable maximum is the number of sockets multiplied by the largest number of ways implemented by any DAWG cache in the system (e.g., 16 or 20). An 8-bit identifier is sufficient to enumerate the maximum active domains even across 8-sockets with 20-way caches.

Importantly, MSR writes to each core's `domain_id`, and each DAWG cache's `policy_hitmap` and `policy_fillmap` MSRs must be a *fence*, prohibiting speculation on these instructions. Failing to do so would permit speculative disabling of DAWG's protection mechanism, leading to Spectre-style vulnerabilities.

D. DAWG's cache eviction/fill isolation

In a simple example of using DAWG at the last level cache (LLC), protection domain `0` (e.g., the kernel) is statically allocated half of DAWG cache's ways, with the other half allocated to unprivileged software (relegated to protection domain `1`). While the cache structure is shared among all software on the system, no access should affect observable cache state across protection domains, considering both the cache data and the metadata. This simple scenario will be generalized to dynamic allocation in Section III-H and we discuss the handling of cache replacement metadata in Section III-J for a variety of replacement policies.

Straightforwardly, cache misses in a DAWG cache must not cause fills or evictions outside the requesting protection domain's ways in order to enforce DAWG's isolation. Like Intel's CAT (Section II-C2), our design ensures that only the ways that a process has been allocated (via its protection domain's `policy_fillmap` policy MSRs) are candidates for eviction; but we also restrict `CLFLUSH` instructions. Hardware instrumentation needed to accomplish this is highlighted in Fig. 4.

E. DAWG's cache metadata isolation

The *cache set metadata* structure in Fig. 4 stores per-line helper data including replacement policy and cache coherence state. The metadata update logic uses tag comparisons (hit information) from all ways to modify set replacement state. DAWG does not leak via the coherence metadata, as coherence traffic is tagged with the requestors's protection domain and does not modify lines in other domains (with a sole exception described in Section III-G).

DAWG's replacement metadata isolation requirement, at a high level, is a non-interference property: victim selection in a protection domain should not be affected by the accesses performed against any other protection domain(s). Furthermore, the cache's replacement policy must allow system software to sanitize the replacement data of a way in order to implement safe protection domain resizing. Details of implementing DAWG-friendly partitionable cache replacement policies are explored in Section III-J.

F. DAWG's cache hit isolation

Cache hits in a DAWG cache must also be isolated, requiring a change to the critical path of the cache structure: a cache access must not hit in ways it was not allocated – a possibility if physical tags are shared across protection domains.

Consider a read access with address $A \implies (T_A, S_A)$ (tag and set, respectively) in a conventional set associative cache. A match on any of the way comparisons indicates a cache

hit ($\exists\ i \mid T_{W_i} == T_A \implies hit$); the associated cache line data is returned to the requesting core, and the replacement policy metadata is updated to make note of the access. This allows a receiver (attacker) to communicate via the cache state by probing the cache tag or metadata state as described in Section II-B.

In DAWG, tag comparisons must be masked with a policy (`policy_hitmap`) that white-lists ways allocated to the requester's protection domain ($\exists\ i \mid$ `policy_hitmap[i]` $\&$ $(T_{W_i} == T_A) \implies hit$). By configuring `policy_hitmap`, system software can ensure cache hits are not visible across protection domains. While the additional required hardware in DAWG caches' hit path adds a gate delay to each cache access, we note that modern L1 caches are usually pipelined. We expect hardware designers will be able to manage an additional low-fanout gate without affecting clock frequency.

In addition to masking hits, DAWG's metadata update must use this policy-masked hit information to modify any replacement policy state safely, preventing information leakage across protection domains via the replacement policy state, as described in Section III-E.

G. Cache lines shared across domains

DAWG effectively hides cache hits outside the white-listed ways as per `policy_hitmap`. While this prevents information leakage via adversarial observation of cached lines, it also complicates the case where addresses are shared across two or more protection domains by allowing ways belonging to different protection domains to have copies of the same line. Read-only data and instruction misses acquire lines in the **S**hared state of the MESI protocol [42] and its variants.

Neither a conventional set associative cache nor Intel's CAT permit duplicating a cache line within a cache: their hardware enforces a simple invariant that *a given tag can only exist in a single way of a cache at any time*. In the case of a DAWG cache, the hardware does not strictly enforce this invariant across protection domains; we allow read-only cache lines (in **S**hared state) to be replicated across ways in different protection domains. Replicating shared cache lines, however, may leak information via the cache coherence protocol (whereby one domain can invalidate lines in another), or violate invariants expected by the cache coherence protocol (by creating a situation where multiple copies of a line exist when one is in the **M**odified state).

In order to maintain isolation, cache coherence traffic must respect DAWG's protection domain boundaries. Requests on the same line from different domains are therefore considered non-matching, and are filled by the memory controller. Cache flush instructions (`CLFLUSH`, `CLWB`) affect only the ways allocated to the requesting `domain_id`. Cross-socket invalidation requests must likewise communicate their originating protection domain. DAWG caches are *not*, however, expected to handle a replicated **M**odified line, meaning system software must not allow shared writable pages across protection domains via a TLB invariant, as described in Section IV-B2.

Stale **S**hared lines of de-allocated pages may linger in the cache; DAWG must invalidate these before zeroing a page to be granted to a process (see Section IV-B2). To this end, DAWG requires a new privileged MSR, with which to *invalidate all copies* of a **S**hared line, given an address, regardless of protection domain. DAWG relies on system software to prevent the case of a replicated **M**odified line.

H. Dynamic allocation of ways

It is unreasonable to implement a static protection domain policy, as it would make inefficient use of the cache resources due to internal fragmentation of ways. Instead, DAWG caches can be provisioned with updated security policies *dynamically*, as the system's workload changes.

In order to maintain its security properties, system software must manage protection domains by manipulating the domains' `policy_hitmap` and `policy_fillmap` MSRs in the DAWG cache. These MSRs are normally equal, but diverge to enable concurrent use of shared caches.

In order to re-assign a DAWG cache way, when creating or modifying the system's protection domains, the way must be *invalidated*, destroying any private information in form of the cache tags and metadata for the way(s) in question. In the case of write-back caches, dirty cache lines in the affected ways must be written-back, or swapped within the set. A privileged software routine flushes one or more ways via a hardware affordance to perform fine-grained cache flushes by set&way, e.g., available on ARM [1].

We require hardware mechanisms to flush a line and/or perform write-back (if **M**), of a specified way in a DAWG memory cache, allowing privileged software to orchestrate way-flushing as part of its software management of protection domains. This functionality is exposed for each cache, and therefore accommodates systems with diverse hierarchies of DAWG caches. We discuss the software mechanism to accommodate dynamic protection domains in Section IV-A2.

While this manuscript does describe the mechanism to adjust the DAWG policies in order to create, grow, or shrink protection domains, we leave as future work resource management support to securely determine the efficient sizes of protection domains for a given workload.

I. Scalability and cache organization

Scalability of the number of active protection domains is a concern with growing number of cores per socket. Since performance critical VMs or containers usually require multiple cores, however, the maximum number of active domains does not have to scale up to the number of cores.

DAWG on non-inclusive LLC caches [25] can also assign zero LLC ways to single-core domains, since these do not need communication via a shared cache. Partitioning must be applied to cache coherence metadata, e.g., snoop filters. Private cache partitioning allows a domain per SMT thread.

On inclusive LLC caches the number of concurrently active domains is limited by the number of ways — for high-core count CPUs this may require increasing associativity, e.g., from

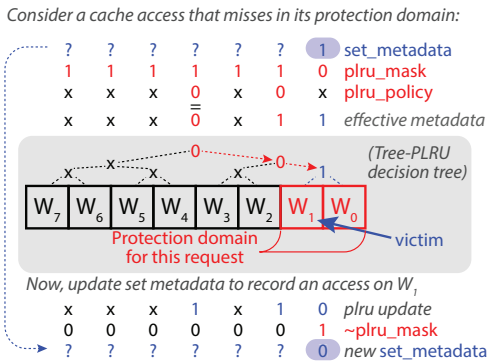

Fig. 5. Victim selection and metadata update with a DAWG-partitioned Tree-PLRU policy.

20-way to 32-way. Partitioning replacement metadata allows high associativity caches with just 1 or 2 bits per tag for metadata to accurately select victims and remain secure.

J. Replacement policies

In this section, we will exemplify the implementation of several common replacement policies compatible with DAWG's isolation requirement. We focus here on several commonplace replacement policies, given that cache replacement policies are diverse. The optimal policy for a workload depends on the effective associativity and may even be software-selected, e.g., ARM A72 [1] allows pseudo-random or pseudo-LRU cache-replacement.

1) Tree-PLRU: pseudo least recently used: Tree-PLRU "approximates" LRU with a small set of bits stored per cache line. The victim selection is a (complete) decision tree informed by metadata bits. `1` signals "go left", whereas `0` signals "go right" to reach the PLRU element, as shown in Fig. 5.

The cache derives `plru_mask` and `plru_policy` from `policy_fillmap`. These fields augment a decision tree over the ways of the cache; a bit of `plru_mask` is `0` if and only if its corresponding subtree in `policy_fillmap` has no zeroes (if the subtree of the decision tree is entirely allocated to the protection domain). Similarly, `plru_policy` bits are set if their corresponding *left* subtrees contain one or more ways allocated to the protection domain. For example, if a protection domain is allocated ways W_0, W_1 of 8 ways, then `plru_mask=0b11111110`, and `plru_policy=0bxxx0x0x` (`0b0000001`, to be precise, with `x` marking masked and unused bits).

At each access, `set_metadata` is updated by changing each bit on the branch leading to the hitting way to be the *opposite* of the direction taken, i.e., "away" from the most recently used way. For example, when accessing W_5, metadata bits are updated by $b_0 \rightarrow 0$, $b_2 \rightarrow 1$, $b_5 \rightarrow 0$. These updates are masked to avoid modifying PLRU bits *above* the allocated subtree. For example, when $\{W_2, W_3\}$ are allocated to the process, and it hits W_3, b_0 and b_1 remain unmodified to avoid leaking information via the metadata updates.

Furthermore, we must mask `set_metadata` bits that are made irrelevant by the allocation. For example, when

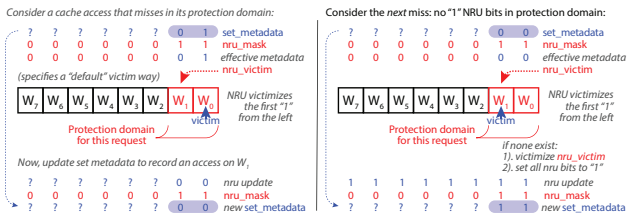

Fig. 6. Victim selection and metadata update with a DAWG-partitioned NRU policy.

$\{W_2, W_3\}$ are allocated to the process, the victim selection should always reach the b_4 node when searching for the pseudo-LRU way. To do this, ignore $\{b_0, b_1\}$ in the metadata table, and use values `0` and `1`, respectively.

Observe that both are straightforwardly implemented via `plru_mask` and `plru_policy`. This forces a subset of decision tree bits, as specified by the policy: victim selection logic uses `( (set_metadata & ~plru_mask) | (plru_mask & plru_policy) )`. This ensures that system software is able to restrict victim selection to a subtree over the cache ways. Metadata updates are partitioned also, by constraining updates to `set_metadata & ~plru_mask`. When system software alters the cache's policies, and re-assigns a way to a different protection domain, it must take care to force the way's metadata to a known value in order to avoid private information leakage.

2) SRRIP and NRU: Not recently used: An NRU policy requires one bit of metadata per way be stored with each set. On a cache hit, the accessed way's NRU bit is set to "0". On a cache miss, the victim is the first (according to some pre-determined order, such as left-to-right) line with a "1" NRU bit. If none exists, the first line is victimized, and all NRU bits of the set are set to "1".

Enforcing DAWG's isolation across protection domains for an NRU policy is a simple matter, as shown in Fig. 6. As before, metadata updates are restricted to the ways white-listed by `nru_mask = policy_fillmap`. In order to victimize only among ways white-listed by the policy, mask the NRU bits of all other ways via `set_metadata & nru_mask` at the input to the NRU replacement logic.

Instead of victimizing the *first* cache line if no "1" bits are found, the victim way must fall into the current protection domain. To implement this, the default victim is specified via `nru_victim`, which selects the leftmost way with a corresponding "1" bit of `nru_mask`, whereas the unmodified NRU is hard-wired to evict a specific way.

The SRRIP [26] replacement policy is similar, but expands the state space of each line from two to four (or more) states by adding a *counter* to track ways less favored to be victimized. Much like NRU, SRRIP victimizes the first (up to some pre-determined order) line with the largest counter during a fill that requires eviction. To partition SRRIP, the same `nru_mask = policy_fillmap` is used, where each line's metadata is masked with the way's bit of `nru_mask` to ensure other domains' lines are considered "recently used" and not

candidates for eviction.

IV. Software Modifications

We describe software provisions for modifying DAWG's protection domains, and also describe small, required modifications to several well-annotated sections of kernel software to implement cross-domain communication primitives robust against speculative execution attacks.

A. Software management of DAWG policies

Protection domains are a software abstraction implemented by system software via DAWG's policy MSRs. The policy MSRs themselves (a table mapping protection `domain_id` to a `policy_hitmap` and `policy_fillmap` at each cache, as described in Section III-B) reside in the DAWG cache hardware, and are atomically modified.

1) DAWG Resource Allocation: Protection domains for a process tree should be specified using the same `cgroup`-like interface as Intel's CAT. In order to orchestrate DAWG's protection domains and policies, the operating system must track the mapping of process IDs to protection domains. In a system with 16 ways in the most associative cache, no more than 16 protection domains can be concurrently scheduled, meaning if the OS has need for more mutually distrusting entities to schedule, it needs to virtualize protection domains by time-multiplexing protection domain IDs, and flushing the ways of the multiplexed domain whenever it is re-allocated.

Another data structure, `dawg_policy`, tracks the resources (cache ways) allocated to each protection domain. This is a table mapping `domain_id` to pairs (`policy_hitmap`, `policy_fillmap`) for each DAWG cache. The kernel uses this table when resizing, creating, or destroying protection domains in order to maintain an exclusive allocation of ways to each protection domain. Whenever one or more ways are re-allocated, the supervisor must look up the current `domain_id` of the owner, accomplished via either a search or a persistent inverse map cache way to `domain_id`.

2) Secure Dynamic Way Reassignment: When modifying an existing allocation of ways in a DAWG cache (writing policy MSRs), as necessary to create or modify protection domains, system software must sanitize (including any replacement metadata, as discussed in Section III-E) the re-allocated way(s) before they may be granted to a new protection domain. The process for re-assigning cache way(s) proceeds as follows:

1) Update the `policy_fillmap` MSRs to disallow fills in the way(s) being transferred out of the shrinking domain.
2) A software loop iterates through the cache's set indexes and flushes all sets of the re-allocated way(s). The shrinking domain may hit on lines yet to be flushed, as `policy_hitmap` is not yet updated.
3) Update the `policy_hitmap` MSRs to exclude ways to be removed from the shrinking protection domain.
4) Update the `policy_hitmap` and `policy_fillmap` MSRs to grant the ways to the growing protection domain.

Higher level policies can be built on this dynamic way-reassignment mechanism.

3) Code Prioritization: Programming the domain selectors for code and data separately allows ways to be dedicated to code without data interference. Commercial studies of code cache sensitivity of production server workloads [25], [27], [41] show large instruction miss rates in L2, but even the largest code working sets fit within 1–2 L3 ways. Code prioritization will also reduce the performance impact of disallowing code sharing across domains, especially when context switching between untrusted domains sharing code.

B. Kernel changes required by DAWG

Consider a likely configuration where a user-mode application and the OS kernel are in different protection domains. In order to perform a system call, communication must occur across the protection domains: the supervisor extracts the (possibly cached) data from the caller by copying into its own memory. In DAWG, this presents a challenge due to strong isolation in the cache.

1) DAWG augments SMAP-annotated sections: We take advantage of modern OS support for the Supervisor Mode Access Prevention (SMAP) feature available in recent x86 architectures, which allows supervisor mode programs to raise a trap on accesses to user-space memory. The intent is to harden the kernel against malicious programs attempting to influence privileged execution via untrusted user-space memory. At each routine where supervisor code intends to access user-space memory, SMAP must be temporarily disabled and subsequently re-enabled via `stac` (Set AC Flag) and `clac` (Clear AC Flag) instructions, respectively. We observe that a modern kernel's interactions with user-space memory are diligently annotated with these instructions, and will refer to these sections as *annotated sections*.

Currently Linux kernels use seven such sections for simple memory copy or clearing routines: `copy_from_user`, `copy_to_user`, `clear_user`, `futex`, etc. We propose extending these annotated sections with short instruction sequences to correctly handle DAWG's communication requirements on system calls and inter-process communication, in addition to the existing handling of the SMAP mechanism. Specifically, sections implementing data movement *from* user to kernel memory are annotated with an MSR write to `domain_id`: *ifetch* and *store* accesses proceed on behalf of the kernel, as before, but *load* accesses use the caller's (user) protection domain. This allows the kernel to efficiently copy from warm cache lines, but preserves isolation. After copying from the user, the `domain_id` MSR is restored to perform all accesses on behalf of the kernel's protection domain. Likewise, sections implementing data movement *to* user memory *ifetch* and *load* on behalf of the kernel's domain, but *store* in the user's cache ways. While the annotated sections may be interrupted by asynchronous events, interrupt handlers are expected to explicitly set `domain_id` to the kernel's protection domain, and restore the MSR to its *prior* state afterwards.

As described in Section III-C, DAWG's `domain_id` MSR writes are a fence, preventing speculative disabling of DAWG's protection mechanism. Current Linux distributions diligently

pair a `stac` instruction with an `lfence` instruction to prevent speculative execution within regions that access user-mode memory, meaning DAWG does not significantly serialize annotated sections over its insecure baseline.

Finally, to guarantee isolation, we require the annotated sections to contain only code that obeys certain properties: to protect against known and future speculative attacks, indirect jumps or calls, and potentially unsafe branches are not to be used. Further, we cannot guarantee that these sections will not require patching as new attacks are discovered, although this is reasonable given the small number and size of the annotated sections.

2) Read-only and CoW sharing across domains: For memory efficiency, DAWG allows securely mapping read-only pages across protection domains, e.g., for shared libraries, requiring hardware cache coherence protocol changes (see Section III-G), and OS/hypervisor support.

This enables conventional system optimizations via page sharing, such as read-only `mmap` from page caches, Copy-on-Write (CoW) conventionally used for `fork`, or for page deduplication across VMs (e.g., Transparent Page Sharing [54]; VM page sharing is typically disabled due to concerns raised by shared cache tag attacks [24]). DAWG maintains security with read-only mappings across protection domains to maintain memory efficiency.

Dirty pages can be prepared for CoW sharing eagerly, or lazily (but cautiously [57]) by installing non-present pages in the consumer domain mapping. Preparing a dirty page for sharing *requires* a write-back of any dirty cache lines on behalf of the producer's domain (via `CLWB` instructions and an appropriate *load* `domain_id`). The writeback guarantees that read-only pages appear only as **S**hared lines in DAWG caches, and can be replicated across protection domains as described in Section III-G.

A write to a page read-only shared across protection domains signals the OS to create a new, private copy using the original producer's `domain_id` for reads, and the consumer's `domain_id` for writes.

3) Reclamation of shared physical pages: Before cache lines may be filled in a new protection domain, pages reclaimed from a protection domain must be removed from DAWG caches as part of normal OS page cleansing. Prior to zeroing (or preparing for DMA) a page previously shared across protection domains, the OS must *invalidate all cache lines* belonging to the page, as described in Section III-G. The same is required between `unmap` and `mmap` operations over the same physical addresses. For most applications, therefore cache line invalidation can be deferred to wholesale destruction of protection domains at `exit`, given ample physical memory.

V. Security Analysis

We explain why DAWG protects against attacks realized thus far on speculative execution processors by stating and arguing a non-interference property in Section V-A. We then argue in Section V-B that system calls and other cross-domain communication are safe. Finally, we show a generalization of our attack schema and point out the limitations of cache partitioning in Section V-C.

A. DAWG Isolation Property

DAWG enforces isolation of *exclusive* protection domains among cache tags and replacement metadata, as long as:

1) victim selection is restricted to the ways allocated to the protection domain (an invariant maintained by system software), and
2) metadata updates as a result of an access in one domain do not affect victim selection in another domain (a requirement on DAWG's cache replacement policy).

Together, this guarantees non-interference – the hits and misses of a program running in one protection domain are unaffected by program behavior in different protection domains. As a result, DAWG blocks the cache tag and metadata channels of non-communicating processes separated by DAWG's protection domains.

B. No leaks from system calls

Consider a case where the kernel (victim) and a user program (attacker) reside in different protection domains. While both use the same cache hierarchy, they share neither cache *lines* nor *metadata* (Section III-B), effectively closing the cache exfiltration channel. In few, well-defined instances where data is passed between them (such as `copy_to_user`), the kernel accesses the attacker's ways to read/write user memory, leaking the (public) access pattern associated with the bulk copying of the syscall inputs and outputs (Section IV-B). Writes to DAWG's MSRs are *fences*, and the annotated sections must not offer opportunity to maliciously mis-speculate control flow (see Section IV-B1), thwarting speculative disabling or misuse of DAWG's protection domains. DAWG also blocks leaks via coherence Metadata, as coherence traffic is restricted to its protection domain (Section III-G), with the sole exception of cross-domain *invalidation*, where physical pages are reclaimed and sanitized.

When re-allocating cache ways, as part of resizing or multiplexing protection domains, no private information is transferred to the receiving protection domain: the kernel sanitizes ways before they are granted, as described in Section IV-A2. Physical pages re-allocated across protection domains are likewise sanitized (Section IV-B3).

When an application makes a system call, the necessary communication (data copying) between kernel and user program must not leak information beyond what is communicated. The OS's correct handling of `domain_id` MSR within annotated sections, as described in Section IV-B ensures user space cache side effects reflect the section's explicit memory accesses.

C. Limitations of cache partitioning

DAWG's cache isolation goals are meant to approach the isolation guarantees of separate machines, yet, even remote network services can fall victim to leaks employing cache tag state for communication. Consider the example of the attacker and victim residing in different protection domains,

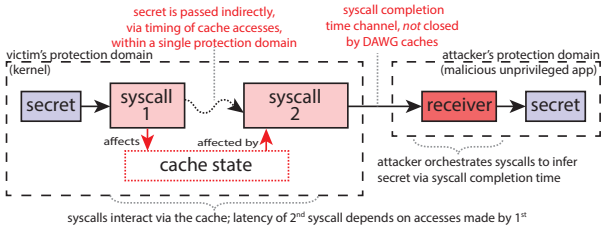

Fig. 7. Generalized Attack Schema: an adversary 1) accesses a victim's secret, 2) reflects it to a transmitter 3) transmits it via a covert channel, 4) receives it in their own protection domain.

sharing no data, but communicating via some API, such as system calls. As in a remote network timing leak [10], where network latency is used to communicate some hidden state in the victim, the *completion time* of API calls can communicate insights about the cache state [31] within a protection domain. Leakage via *reflection* through the cache is thus possible: the receiver invokes an API call that accesses private information, which affects the state of its private cache ways. The receiver then exfiltrates this information via the latency of another API call. Fig. 7 shows a cache timing leak which relies on cache reflection entirely within the victim's protection domain. The syscall completion time channel is used for exfiltration, meaning no private information crosses DAWG's domain boundaries in the caches, rendering DAWG, and cache partitioning in general, ineffective at closing a leak of this type.

The transmitter is instructed via an API call to access $a[b[i]]$, where i is provided by the receiver (via `syscall1`), while a, b reside in the victim's protection domain. The cache tag state of the transmitter now reflects $b[i]$, affecting the latency of subsequent syscalls in a way dependent on the secret $b[i]$. The receiver now exfiltrates information about $b[i]$ by selecting a j from the space of possible values of $b[i]$ and measuring the completion time of `syscall2`, which accesses $a[j]$. The syscall completion time communicates whether the transmitter hits on $a[j]$, which implies $a[j] \cong a[b[i]]$, and for a compact a, that $b[i] = j$ – a leak. This leak can be amplified by initializing cache state via a bulk memory operation, and, for a machine-local receiver by malicious mis-speculation.

While not the focus of this paper, for completeness, we outline a few countermeasures for this type of leak. Observe that the completion time of a public API is used here to exfiltrate private information. The execution time of a syscall can be padded to guarantee constant (and worst-case) latency, no matter the input or internal state. This can be relaxed to bound the leak to a known number of bits per access [17].

A zero leak countermeasure requires destroying the transmitting domain's cache state across syscalls/API invocations, preventing reflection via the cache. DAWG can make this less inefficient: in addition to dynamic resizing, setting the replacement mask `policy_fillmap` to a subset of the `policy_hitmap` allows locking cache ways to preserve the hot working set. This ensures that all unique cache lines accessed during one request have constant observable time.

TABLE I
SIMULATED SYSTEM SPECIFICATIONS.

Cores		DRAM	Bandwidth
Count	Frequency	Controllers	*Peak*
8 OoO	3 GHz	4 x DDR3-1333	42 GB/s

Private Caches			Shared Cache	
L1	L2	Organization	L3	Organization
2× 32 KB	256 KB	8-way PLRU	8× 2 MB	16-way NRU

VI. EVALUATION

To evaluate DAWG, we use the `zsim` [48] execution-driven x86-64 simulator and Haswell hardware [15] for our experiments.

A. Configuration of insecure baseline

Table I summarizes the characteristics of the simulated environment. The out-of-order model implemented by `zsim` is calibrated against Intel Westmere, informing our choice of cache and network-on-chip latencies. The DRAM configuration is typical for contemporary servers at ∼5 GB/s theoretical DRAM bandwidth per core. Our baseline uses the Tree-PLRU (Section III-J1) replacement policy for private caches, and a 2-bit NRU for the shared LLC. The simulated model implements inclusive caches, although DAWG domains with reduced associativity would benefit from relaxed inclusion [25]. We simulate CAT partitioning at *all* levels of the cache, while modern hardware only offers this at the LLC. We do this by restricting the replacement mask `policy_fillmap`, while white-listing all ways via the `policy_hitmap`.

B. DAWG Policy Scenarios

We evaluate several protection domain configurations for different resource sharing and isolation scenarios.

1) VM or container isolation on dedicated cores: Isolating peer protection domains from one another requires equitable LLC partitioning, e.g., 50% of ways allocated to two active domains. In the case of cores dedicated to each workload (no context switches), each scheduled domain is assigned the entirety of its L1 and L2.

2) VM or container isolation on time-shared cores: To allow the OS to overcommit cores across protection domains (thus requiring frequent context switches between domains), we also evaluate a partitioned L2 cache.

3) OS isolation: Only two DAWG domains are needed to isolate an OS from applications. For processes with few OS interventions in the steady state, e.g., SPECCPU workloads, the OS can reserve a single way in the LLC, and flush L1 and L2 ways to service the rare system calls. Processes utilizing more OS services would benefit from more ways allocated to OS's domain.

C. DAWG versus insecure baseline

Way partitioning mechanisms reduce cache capacity and associativity, which increases conflict misses, but improves fairness and reduces contention. We refer to CAT [21] for analysis of the performance impact of way partitioning on a

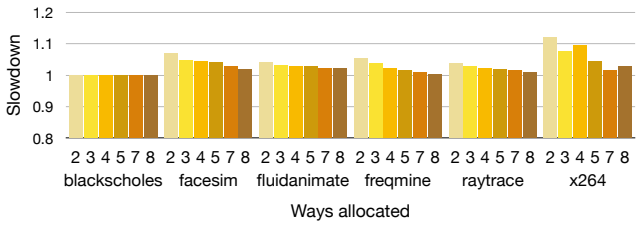

Fig. 8. Way partitioning performance at low associativity in all caches (8-way L1, 8-way L2, and 16-way L3).

subset of SPEC CPU2006. Here, we evaluate CAT and DAWG on parallel applications from PARSEC [6], and parallel graph applications from the GAP Benchmark Suite (GAPBS) [4], which allows a sweep of of workload sizes.

Fig. 8 shows DAWG partitioning of private L1 and L2 (Section VI-B2) caches in addition to the L3. We explore DAWG configurations on a subset of PARSEC benchmarks on `simlarge` workloads. The cache insensitive `blackscholes` (or omitted `swaptions` with 0.001 L2 MPKI (Misses Per 1000 Instructions)) are unaffected at any way allocation. For a VM isolation policy (Section VI-B1) with $^8/_{16}$ of the L3, even workloads with higher MPKI such as `facesim` show at most 2% slowdown. The $\langle ^2/_8$ L2, $^2/_{16}$ L3$\rangle$ configuration is affected by both capacity and associativity reductions, yet most benchmarks have 4–7% slowdown, up to 12% for x264. Such an extreme configuration can accommodate 4 very frequently context switched protection domains.

Fig. 9 shows the performance of protection domains using different fractions of an L3 cache for 4-thread instances of graph applications from GAPBS. We use variable size synthetic *power law* graphs [12], [18] that match the structure of real-world social and web graphs and therefore exhibit cache locality [5]. The power law structure, however, implies that there is diminishing return from each additional L3 way. As shown, at half cache capacity ($^8/_{16}$ L3, Section VI-B1), there is at most 15% slowdown (`bc` and `tc` benchmarks) at the largest simulated size (2^{20} vertices). A characteristic *eye* is formed when the performance curves of different configurations cross over the working set boundary (e.g., graph size of 2^{17}). Performance with working sets smaller or larger than the effective cache capacity is unaffected — at the largest size `cc`, `pr`, and `sssp` show 1–4% slowdown.

Reserving for the OS (Section VI-B3), one way (6% of LLC capacity) adds no performance overhead to most workloads. The only exception would be a workload caught in the *eye*, e.g., PageRank at 2^{17} has 30% overhead (Fig. 9), while at 2^{16} or 2^{18} — 0% difference.

D. CAT versus DAWG

We analyze and evaluate scenarios based on the degree of code and data sharing across domains.

1) No Sharing: There is virtually no performance difference between secure DAWG partitioning, and insecure CAT partitioning in the absence of read-sharing across domains.

DAWG reduces interference in replacement metadata updates and enforces the intended replacement strategy within a domain,

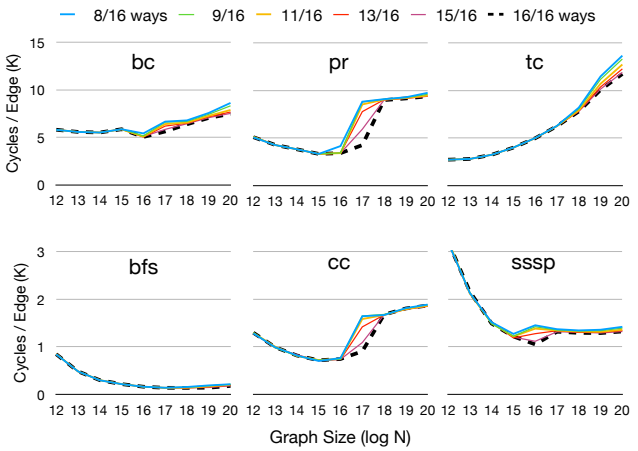

Fig. 9. Way partitioning performance with varying working set on graph applications. Simulated 16-way L3.

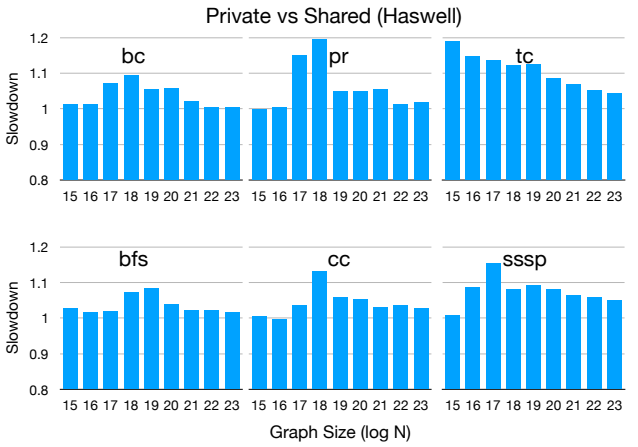

Fig. 10. Read-only sharing effects of two instances using Shared vs Private data of varying scale (1-thread instances). Actual Haswell 20-way 30 MB L3.

while CAT may lose block history effectively exhibiting random replacement – a minor, workload-dependent perturbation. In simulations (not shown), we replicate a known observation that random replacement occasionally performs better than LRU near cache capacity. We did not observe this effect with NRU replacement.

2) Read-only Sharing: CAT QoS guarantees a lower bound on a workload's effective cache capacity, while DAWG isolation forces a tight upper bound. DAWG's isolation reduces cache capacity compared to CAT when cache lines are read-only shared across mutually untrusting protection domains. CAT permits hits across partitions where code or read-only data are unsafely shared. We focus on read-only data in our evaluation, as benchmarks with low L1i MPKI like GAPBS, PARSEC, or SPECCPU are poorly suited to study code cache sensitivity.

We analyze real applications using one line modifications

to GAPBS to `fork` (a single-thread process) either before or after creating in-memory graph representations. The first results in a private graph for each process, while the latter simulates `mmap` of a shared graph. The shared graphs access read-only data across domains in the baseline and CAT, while DAWG has to replicate data in domain-private ways. Since `zsim` does not simulate TLBs, we ensure different virtual addresses are used to avoid false sharing. We first verified in simulation that DAWG, with memory shared across protection domains, behaves identically to CAT and the baseline with private data.

Next, we demonstrate (in Fig. 10) that these benchmarks show little performance difference on real hardware [15] for most data sizes; Shared baseline models Shared CAT, while Private baseline models Shared DAWG. The majority of cycles are spent on random accesses to read-write data, while read-only data is streamed sequentially. Although read-only data is much larger than read-write data (e.g., 16 times more edges than vertices), prefetching and scan- and thrash- resistant policies [26], [45] further reduce the need for cache resident read-only data. Note that even at 2^{23} vertices these effects are immaterial; real-world graphs have billions of people or pages.

E. Domain copy microbenchmark

We simulated a privilege level change at simulated system calls for user-mode TCP/IP. Since `copy_from_user` and `copy_to_user` permit hits in the producer's ways, there is no performance difference against the baseline (not shown).

VII. Conclusion

DAWG protects against attacks that rely on a cache state-based channel, which are commonly referred to as cache-timing attacks, on speculative execution processors with reasonable overheads. The same policies can be applied to any set-associative structure, e.g., TLB or branch history tables. DAWG has its limitations and additional techniques are required to block exfiltration channels different from the cache channel. We believe that techniques like DAWG are needed to restore our confidence in public cloud infrastructure, and hardware and software co-design will help minimize performance overheads.

A good proxy for the performance overheads of secure DAWG is Intel's existing, though insecure, CAT hardware. Traditional QoS uses of CAT, however, differ from desired DAWG protection domains' configurations. Research on software resource management strategies can therefore commence with evaluation of large scale workloads on CAT. CPU vendors can similarly analyze the cost-benefits of increasing cache capacity and associativity to accommodate larger numbers of active protection domains.

VIII. Acknowledgments

We are grateful to Carl Waldspurger for his valuable feedback on the initial design as well as the final presentation of this paper. We also thank our anonymous reviewers and Julian Shun for helpful questions and comments.

References

[1] ARM, "ARM Cortex-A72 MPCore processor technical reference manual," 2015.
[2] ARM, "ARM Software Speculation Barrier," https://github.com/ARM-software/speculation-barrier, January 2018.
[3] S. Banescu, "Cache timing attacks," 2011, [Online; accessed 26-January-2014].
[4] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619
[5] ——, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, 2015, pp. 56–65.
[6] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[7] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006, pp. 201–215.
[8] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security–ESORICS*. Springer, 2011.
[9] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, 2005.
[10] ——, "Remote timing attacks are practical," *Computer Networks*, 2005.
[11] C. Carruth, "Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities," http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html, January 2018.
[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, 2004, pp. 442–446.
[13] J. Corbet, "KAISER: hiding the kernel from user space," https://lwn.net/Articles/738975/, November 2017.
[14] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *Transactions on Architecture and Code Optimization (TACO)*, 2012.
[15] E5v3, "Intel Xeon Processor E5-2680 v3(30M Cache, 2.50 GHz)," http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz.
[16] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of the 24th international symposium on High Performance Computer Architecture (HPCA-24)*, February 2018.
[17] C. W. Fletcher, L. Ren, X. Yu, M. V. Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 213–224.
[18] Graph500, "Graph 500 benchmark," http://www.graph500.org/specifications.
[19] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 217–233. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss
[20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.
[21] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 657–668.
[22] J. Horn, "Reading privileged memory with a side-channel," https://googleprojectzero.blogspot.com/2018/01/, January 2018.
[23] Intel Corp., "Improving real-time performance by utilizing Cache Allocation Technology," April 2015.

[24] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-VM attack on AES," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[25] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 343–353.

[26] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, 2010, pp. 60–71. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815971

[27] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 158–169.

[28] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[29] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *Transactions on Computer Systems (TOCS)*, 1992.

[30] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *ArXiv e-prints*, Jul. 2018.

[31] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[32] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology (CRYPTO)*. Springer, 1996.

[33] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *workshop on Computer security architectures*. ACM, 2008.

[34] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA*. IEEE, 2008.

[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[36] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, Mar 2016.

[37] F. Liu and R. B. Lee, "Random fill cache architecture," in *Microarchitecture (MICRO)*. IEEE, 2014.

[38] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy*. IEEE, 2015.

[39] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox – practical cache attacks in javascript," *arXiv preprint arXiv:1502.07373*, 2015.

[40] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology–CT-RSA 2006*. Springer, 2006, pp. 1–20.

[41] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2017, pp. 233–244.

[42] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984.

[43] A. Pardoe, "Spectre mitigations in MSVC," https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, January 2018.

[44] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830773

[45] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. S. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250709

[46] Richard Grisenthwaite, "Cache Speculation Side-channels," January 2018.

[47] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 57–68.

[48] ——, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture-ISCA*, vol. 13. Association for Computing Machinery, 2013, pp. 23–27.

[49] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. H. Campbell, "A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds," *CoRR*, vol. abs/1708.09538, 2017. [Online]. Available: http://arxiv.org/abs/1708.09538

[50] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice - a low-cost high-speed address translation mechanism," *SIGARCH Computer Architecture News*, 1990.

[51] L. Torvalds, "Re: Page colouring," 2003. [Online]. Available: http://yarchive.net/comp/linux/cache_coloring.html

[52] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *arXiv preprint arXiv:1802.03802*, 2018.

[53] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886, January 2018.

[54] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, ser. OSDI '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 181–194. [Online]. Available: http://dl.acm.org/citation.cfm?id=1060289.1060307

[55] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 74:1–74:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898086

[56] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[57] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 640–656.

[58] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 347–360. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080222

[59] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 168–179.

[60] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack." in *USENIX Security Symposium*, 2014.

[61] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 89–102. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519076