Cache-related Hardware Capabilities and Their Impact on Information Security

RODRIGO BRANCO and BEN LEE, Oregon State University, School of Electrical Engineering and Computer Science

Caching is an important technique to speed-up execution, and its implementation and use cases vary. When applied specifically to the memory hierarchy, caching is used to speed up memory accesses and memory translations. Different cache implementations are considered microarchitectural secrets and oftentimes change between generations. The integration of caches in hardware greatly influences security policy enforcement in the platform since caches maintain copies of code and data and their security properties. Examples of attacks due to the existence of caches are side-channels against cryptographic software, recent speculative execution abuses to leak secret data, and usages of cache-based manipulations (e.g., forcing cache splits/incoherence) to hide from security software detection. This survey examines the security issues due to different cache usages in a microarchitecture. The survey also explains the most complicated caching features and their impact on the security of the platform in different scenarios.

CCS Concepts: • **Security and privacy → Security in hardware**; **Hardware attacks and countermeasures**; **Side-channel analysis and countermeasures;**

Additional Key Words and Phrases: Hardware cache, memory layout, cache-related attacks

ACM Reference format:
Rodrigo Branco and Ben Lee. 2022. Cache-related Hardware Capabilities and Their Impact on Information Security. *ACM Comput. Surv.* 55, 6, Article 125 (December 2022), 35 pages.
https://doi.org/10.1145/3534962

1 INTRODUCTION

Processing efficiency of modern computing platforms has improved significantly to provide more computing power. However, one of the main challenges is the performance gap between smaller but faster memories such as registers/caches and the bigger but slower memories such as DRAM and secondary storage, i.e., hard disk drives and **Solid State Drives** (**SSDs**). In order to address the CPU-Memory performance gap, modern computers provide caching capabilities that have evolved quickly and substantially affect the way data are transferred between the CPU and the system memory. Although cache structures have changed across different generations of a platform, most of those changes are transparent to software (i.e., no software changes are required) because caches are considered a microarchitectural feature as opposed to an architectural one. As such, caches are typically used as an intermediary storage between the system memory and CPU cores. However, there are also less obvious cases of cache usage involving branch prediction, page translation, and

Authors' address: R. Branco and B. Lee, Oregon State University, School of Electrical Engineering and Computer Science; emails: rodrigo@kernelhacking.com, rodrigofws@gmail.com, benl@engr.orst.edu.

0360-0300/2022/12-ART125 $15.00
https://doi.org/10.1145/3534962

caching the history of decoded instructions (e.g., μop cache [46]). Therefore, in this article, the term *cache* is used to describe any structure that holds a copy of a data to optimize its access time. More importantly, a cache is a shared resource that represents a potential source of information leakage in the platform.

Understanding cache-related vulnerabilities is challenging because a general categorization of caching mechanisms in the context of security does not exist. Some of the proposed attacks in the literature give specifics about cache inner workings [66, 68], while others only provide a high-level description of the characteristics that influence the discussed attacks [47, 75] or focus exclusively on surveying the side-channels and mitigations that become quickly outdated [32, 52]. However, none of them provide a holistic view of cache components in a system involving various types of cache-related attacks discussed in this article. Therefore, this article analyzes caches in a platform from a security pointofview to illustrate their impact on security promises and assumptions in software as well as surveys security weaknesses that exist because of such cache elements. This article covers the latest speculative-related covert-channels, introduces a new classification based on the attack objective, and details specific implementation features that are relevant to the examples that are discussed. Therefore, the objective is to explain the cache internals in a general way, but pinpoint specifics when the differences matter in terms of security. While the focus of the discussion is based on Intel-based architectures and microarchitectures, many of the concepts and issues discussed apply to other architectures as well.

The survey is organized as follows: Section 2 presents the background information on cache-related security considerations for both attack/defense and threat modeling, including a taxonomy to help understand the different security aspects that need to be considered when a cache is included in a system. Section 3 introduces different concepts related to caches and cache configurations in a modern x86-64 microarchitecture. These concepts will help readers better understand the complexity and versatility of current usages of caches surveyed in Section 4. Moreover, Section 4 details security concerns using specific scenarios as well as their impact and mitigations. Finally, Section 5 concludes this survey.[1]

2 BACKGROUND ON CACHE-BASED ATTACKS AND DEFENSES

Caches are important to security for the following reasons:

- Caches are frequently shared between entities with different access permissions, which affect security policies (e.g., software running in different privilege modes, such as rings 0 and 3) and contexts (e.g., system management mode and hypervisor);
- since caches are on the direct path of data accesses, if they are not properly updated with the current data being used, they will provide the wrong view of the system state. For example, an I/O device reads the system memory via DMA and if the data are not flushed from the cache to the system memory or the platform does not provide coherency between the cache and the system memory for those reads, the I/O device will see different data than the one in the cache; and
- caches influence the access speed with the side-effect of potentially leaking undesired information, i.e., creating a side-channel that leaks key information used for encryption.

The formal definition of caches is well understood, but the composition of caches within the entire system and their impact on security is a fairly new area. A modern computer system provides

[1]The technical terminology used in the survey is based on the perspective of the hardware, as such it does not differentiate, for example, between page and segmentation faults.

many policies for an operating system to protect/isolate different programs running on the system, such as separate address spaces and different access permissions. Since a cache holds copies of data that are used by the platform, including for internal operations (i.e., branch prediction), it must be treated as a policy enforcement point. In other words, if an application accesses information from the cache without having the proper context regarding the intended protection (i.e., address space/memory permissions), then the policy cannot be enforced. Cache operations such as flushing a given cache block using the `clflush` instruction or executing a sequence of branches that adds entries to the branch predictor tables can be considered as primitives or capabilities available to an attacker. Therefore, understanding the potential or unintended security impact of using such primitives is essential. For example, if an attacker is able to observe the performance improvement due to cached data, this may expose specific code decisions that were not meant to be unveiled. This is referred to as a *cache-based side-channel*.

Security problems presented in the literature are usually focused only on the cache characteristics that apply to a specific problem without considering other important cache factors, such as those described in Section 3 that cover a variety of configuration options that change how caches behave. In addition, the existence of different caches is oftentimes ignored when performing security impact analysis. This makes it much harder to (1) understand the impact of caches on security; (2) design solutions that are resistant to potential security problems; and (3) leverage cache characteristics to implement security protections for new attacks.

The following subsections present a threat model for caches in a system and discuss how this serves as a taxonomy for cache security analysis in both cache-based attacks and defenses.

2.1 Threat Modeling

Threat modeling is the process of understanding the potential threats to a given system (software, hardware, or the entire platform) [67]. A threat model can be created for an entire platform, a part of the system, or a specific component. The process starts by defining assets (i.e., what needs to be protected) and security objectives (i.e., what security guarantees the system provides to its assets). An example of a security objective is protecting against physical attacks. A ***trusted computing base*** (**TCB**) is also defined, which is essentially a set of all the elements that are trusted (i.e., these are not considered as attacking entities) by the system to meet the security objectives. Anything that is considered to be a part of the TCB must also meet the security objectives because, if it is compromised, it would also affect the element for which the threat model was defined. This is the reason why it is difficult to combine different threat models for different components to create a single one for an entire system. Threats or *adversaries* are also identified as they are the potential points of origin for attacks. These can be anything that is untrusted for a specific system and need to be mitigated according to the security objectives, e.g., other devices in the platform, software running with high privileges, and so on. Caches as a shared resource change the expectations of different components in the system potentially causing situations where the security objectives are broken (e.g., secrets leaked due to a cache-based side-channel). This means that a threat model for any component in a platform must include the different caches and how they impact the security objectives of the entire system.

2.2 Cache-based Attacks and Security Vulnerability Taxonomy for Cache-based Issues

This subsection analyzes the weaknesses that arise from having a cache and the attack objectives in exploiting such weaknesses. A taxonomy on security vulnerabilities is also defined based on the following five types of attacks that can occur due to the presence of a cache:

(1) Type 1—Creating an *asymmetry* (also called a split or an async) between the intended data to be cached and the actual data cached. An asymmetry can occur between two different

caches or a cache and DRAM, which are supposed to hold the same data but are on two different data paths, i.e., used by different components and at different times. This can lead to bypassing of security functionalities that verify the integrity of data;
(2) Type 2—Leaking of information due to *side-channels*;
(3) Type 3—Breaking isolation policies on the platform by transferring data using the cache, which is referred to as a *covert-channel*;
(4) Type 4—*Cache access denial*, which would incur a significant performance penalty. The denial may be *partial* (i.e., the attacker manages to make the cache unavailable for certain parts of a target application) or *permanent* (i.e., the attacker manages to make the cache completely unavailable to a target application); and
(5) Type 5—*Wrongly cached value* (or invalid cache hit), which occurs when a cache hit happens in the wrong context.

In Type 1 attacks, an attacker's intention is to either leverage some asymmetry among different caches or create a difference between the cache data and the system memory data with the objective of circumventing a security technology. An example of the former case is having different entries for the same memory address in the data and instruction **Translation Lookaside Buffers** (**TLBs**), as explained in Section 4.1. An example of the latter case is a software-based or DMA-based memory acquisition device used for forensic analysis [61]. Other hardware-based attacks against memory acquisition, such as using the**Input/Output Memory Management Unit** (**IOMMU**) to block memory accesses [77], are beyond the scope of this survey since they do not involve the usage of caches [61].

Type 2 attacks can infer the actual computing paths taken by an algorithm because a cache hit has a higher performance than a cache miss. Moreover, if the decision paths involve key-related information, parts of the key can be discovered. There are many existing works that demonstrate the viability of cache timing and probe related side-channel attacks [50, 83].

Type 3 attacks target the shared cache properties to bypass security policies and transfer information using the cache as a covert-channel, even when such a transfer is prohibited [79]. Many of the speculative execution attacks (also known as *transient execution attacks*) leverage the control over the speculation of a target application to execute out-of-order code, called a gadget, which transfers data using the cache [80]. A requirement for such attacks is the presence of **Out-of-Order** (**OoO**) execution.

Type 4 attacks represent situations where access to a cache are denied causing a slow-down. Note that this is not really a **denial-of-service** (**DoS**) per-se since the cache is an optimization feature. Cache access denial or less than ideal cache usage scenario is usually an outcome of mitigation proposals for other attacks (e.g., less cache is available due to splitting or partitioning), which ultimately slows down the platform. This type also covers attacks such as memory hog applications that leverage the way the memory is accessed. For example, the memory controller has a buffer that retains the last accessed memory row to optimize memory access and such a row buffer is essentially a cache. A memory hog application can repeatedly accesses the same memory row to have a priority in having its requests met and in the process slow down other applications in the system [56]. This is not a full DoS because the other requests will wait for a certain period of time in a priority queue and then their priorities are raised and the applications will get serviced, but nevertheless the they are slowed-down. This problem is still serious, especially for cloud-based systems shared by multiple entities.

Type 5 attacks return incorrect information. An example of such a case is when the system is running in unprivileged mode (ring 3) and a hit on the micro-operation (μop) cache returns μops

that were cached during privileged mode execution (ring 0). If the returned μops have different semantics for privileged versus unprivileged modes, unexpected behavior can occur. An example of this problem is discussed in Section 4.6.

In [32], the authors defined a taxonomy based on the sharing level of the cache (thread or system shared) and the degree of temporal concurrency between the exploit and the target application. Such a taxonomy is excellent when all instances of a given vulnerability class (side-channels for the case of [32]) are enumerated. Another survey also proposed a taxonomy, but the focus was on evaluating cryptographic implementations and their resistance to side-channels [52]. In contrast, this survey develops a taxonomy based on the exploited weaknesses to provide a full understanding of all possible classes of security issues that occur in any cache implementation including future ones. The classes of cache-based attacks will expand as technology evolves, but hopefully, the overall classification method proposed in this survey will still remain relevant.

3 CACHE AND MICROARCHITECTURE PRINCIPLES

This section discusses some cache principles necessary to understand the cache characteristics that attackers can exploit. While many of the principles discussed are generally well known, this section also discusses how different configurations can cause certain behaviors in modern platforms that are not officially documented.

3.1 No-fill and Non-eviction Modes

The idea behind the *no-fill mode* is that no new entries are created, but write hits still update the cache. As such, the no-fill mode can be used to create cache splitting where the cache content is different than the DRAM content. If an encryption key is stored in the cache, but not in DRAM (i.e., the DRAM contents are erased), a Cold Boot attack will not be able to access the key (see Online Supplement). But since accesses still hit the cache, the key is still available for use within the same machine.

On the other hand, the *non-eviction mode* guarantees that data in the cache will not be evicted, which is an obscure feature and is not documented in publicly available Intel architecture manuals, but is mentioned in less known materials [11, 12, 24]. This mode is used by BIOS [55] during the early boot process to enable **cache-as-RAM** (**CAR**). CAR is a way to execute code that supposedly needs memory (e.g., function calls require memory since they use the stack) in early boot phases when memory is still not available.

3.2 Influence of Direct Memory Access (DMA) on Caches

On Intel platforms, **Directory Memory Access** (**DMA**) is for the most part coherent [69]. This means that if a DMA is configured to read a range of memory addresses, and if the cache holds the most up-to-date information, the cached data will be returned. Figure 1 illustrates how DMA operates, based on the explanation provided in [6].

In Figure 1, a PCIe device issues a DMA access to the PCIe CPU Agent (1), which is then forwarded to the DMA Arbitrator (2a and 2b). Although only one request is sent to the arbitrator, it behaves as if two requests were made – one for the memory (path 2a) and the other for the cache contents (path 2b). The DMA Arbitrator then generates two requests: one for the CPU/caches (3a) and the other for the **memory controller** (**MC**) (3b). If there is a cache hit, the DMA Arbitrator receives a response from the cache with the data (4a). The memory controller also sends the request to the system memory and it responds back to the DMA Arbitrator (4b). Finally, the DMA Arbitrator sends back the most up-to-date data, either from the cache or the system memory.

Given the DMA coherency guarantees in the x86-64 platform, if a device uses DMA to access memory (see Section 2.2 for discussion on Type 1 attacks) in any modern Intel microarchitecture,

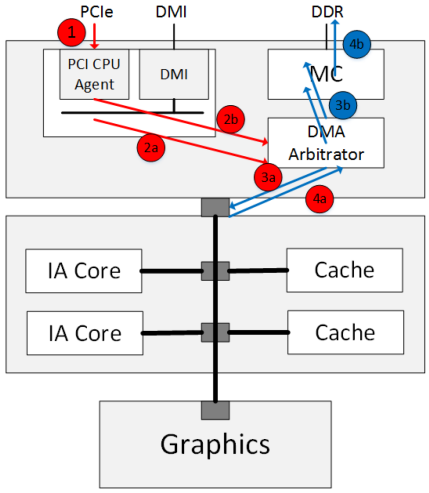


Fig. 1. DMA access and cache coherency.

it will return the most recent data between the cache and the system memory. But, this will not be the case in other microarchitectures that do not guarantee cache coherence. Note that this is a design decision, not a security weakness per-se, unless there is a dependency (assumption) on the behavior. An example of such a case would be a forensics device that uses DMA to capture DRAM contents for offline examination.[2]

3.3 Non-coherent TLBs

The **Memory Management Unit** (**MMU**) is a hardware component that performs virtual-to-physical address translations and is usually integrated in the processor. The MMU performs a page walk (see Section 3.4) by traversing multiple levels of page tables to find the physical address for a given virtual address and caches the result in the TLB.[3] Since instruction fetches and data accesses are different, each CPU core implements at least two different TLBs—one for **instructions TLB** (**I-TLB**) and the other for **data TLB** (**D-TLB**). Additionally, modern platforms also have a Second-level or **Shared TLB** (**S-TLB**). The S-TLB is a unified cache that boosts the performance of I-TLB and D-TLB accesses by increasing their sizes as a level-2 TLB. The S-TLB keeps only a single entry for a linear address and then invalidates the entries in D-TLB and I-TLB, which is referred to as *merging*. Note that TLB is a general term that reflects the hierarchy of the different, specialized TLBs and should not be confused with the specific I-TLB, D-TLB, or S-TLB.

Each page in a system is referenced by a datastructure that has different configuration bits (i.e., present bit, dirty bit, user/supervisor). But not all of the configuration bits are reflected in the TLBs and which ones are included are not officially documented (with some mentioned in [13]). An important configuration bit that is included in the TLB is the ***eXecution Disable*** (**XD**) (also known as ***Non-eXecutable*** (**NX**)) a bit because in principle non-executable pages should not be in the instruction cache. But, if an instruction fetch generates a fault because the NX bit is set, having the bit in the I-TLB means that the translation can still be cached. If another fault occurs at the same instruction, a page walk becomes unnecessary because that entry in the I-TLB has its NX bit set.

[2]DMA access technologies for servers have special behavior and new optimizations exist, such as **Data Direct I/O** (**DDIO**) [21] which was recently shown to be vulnerable to side-channels [49].
[3]The TLB uses linear address (after segmentation) and not logical address (before segmentation).

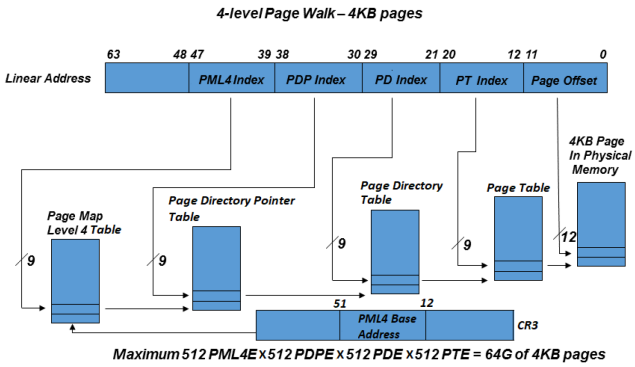


Fig. 2. Mapping of 4 KB pages. CR3 points to the PML4 table.

Unlike DMA accesses, accessing the TLBs does not guarantee coherency in multi-core systems. For example, consider a system with two cores (Cores 1 and 2), where each core has its own I-TLB, D-TLB, and S-TLB. If Core 1 loads an I-TLB entry for an executable page meaning an instruction from that page was executed, and later the page is marked in memory as non-executable, there will be no automatic synchronization between I-TLB and S-TLB.[4] This means the configuration for a page in memory and its corresponding entry in the TLB hierarchy do not match at this point; therefore, a code fetch that eventually hits that I-TLB entry will not generate a fault.[5] In addition, TLB entries for non-executable pages are not cached in the I-TLB, although they may be cached in the S-TLB.

Since any changes to page table entries are made by the operating system and the non-coherent behavior of TLBs is known (i.e., documented), no issues will arise from a well-behaving (non-buggy) operating system. However, a malicious entity, such as a rootkit, is able to create a split where a page is marked as executable in the TLB but its page table entry is marked as non-executable. If a security software only expects to locate malicious code in pages marked as executable, it will be completely blind to the malicious code that exists in non-executable pages. Other cases of security software bypasses using cache split are discussed in Section 4.1.

3.4 Physical Address Extension and Page Walk

Paging in a modern x86 system is quite complicated with different options available to the operating system [13]. There are four modes of page lookup:

(1) 32-bit paging;
(2) PAE paging;
(3) 4-level paging; and
(4) 5-level paging.

Figure 2 shows a page walk process and how 4 KB pages are mapped. In the figure, the linear address contains four 9-bit index fields plus a 12-bit page offset. First, the address in the ***Control Register 3*** (**CR3**) points to the ***Page Map Level 4*** (**PML4**) table. CR3 together with the *PML4 Index* is used to point to a ***PML4 Entry*** (**PML4E**). This is then used together with the ***Page Directory Pointer*** (**PDP**) *Index* to point to a ***Page Directory Pointer Entry*** (**PDPE**). The PDPE is used together with the *Page Directory Index* to point to a ***Page Directory Entry*** (**PDE**). The PDE together

[4]Note that the entry in S-TLB might also exist because the loading of the program to memory cached it.
[5]Note that some microarchitectures do implement a re-walk in the case of a fault, thereby forcing synchronization.

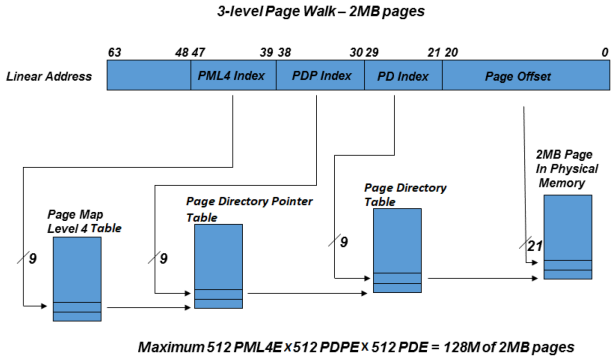


Fig. 3. Mapping of 2 MB pages. CR3 points to the PML4 table.

with the *Page Table Index* points to a ***Page Table Entry*** (**PTE**). Finally, PTE together with the *Page Offset* points to a 4 KB physical page. A system with 2 MB pages can be implemented using a 3-level page walk by combining the 9-bit Page Table Index with the 12-bit Page Offset into a 21-bit *Page Offset* field as shown in Figure 3. Note that 2 MB pages can also be mixed and matched with 4 KB pages.

The ***Control Register 4*** (**CR4**) is used to control several architectural extensions, many of which influence the caches on the platform. The bits in *CR4* that are most relevant to this discussion are the ***Physical Address Extension*** (**PAE**) bit and the ***Page Size Extension*** (**PSE**) bit. The PAE bit is used to define the type of page walk to be performed, and extends paging structures to support 64-bit addresses allowing for at least 36-bit physical address space.[6] On the other hand, the PSE bit enables the support for pages larger than 4 KB. Each PDE also includes the ***Page Size*** (**PS**) bit that determines the page size. The ***Extended Feature Enable Register, Long Mode Active*** (**EFER.LMA**) bit, which is accessible through a **Model Specific Register** (**MSR**) in Intel platforms (MSR 0xC0000080), also affects paging and was added to support entering and exiting the long mode (i.e., 64-bit support together with PAE and PG). As a side note, the CR4 register also has the ***Page Global Enable*** (**PGE**) bit, which allows the usage of the ***global*** (***G***) bit in PTE to prevent the page controlled by the PTE from being flushed on writes to CR3.

Combining all the possible configurations leads to at least seven types of page walks.[7] This includes four different modes of page lookup and different CPU addressing modes. Table 1 lists all the possible configurations in a system. In addition, the usage of ***Extended Page Tables*** (**EPT**), which improves **Virtualization Extensions** (**VT-x**) by providing hardware support for virtualization, changes the results of any of the existing modes. This makes it possible to revive an old attack called *Shadow Walker*, which is analyzed in Section 4.1. The EPT mode is orthogonal to the existing modes and largely resembles the 48-bit Linear Address Space Mode. The challenge in protecting a system or defining the impact of a new attack technique is that a specific configuration changes the overall behavior and may make attacks not possible, or worse, they might endup allowing attacks.

Since there are several types of page walks, the impact caches have on different scenarios depends on the configuration of the platform. In other words, different operating systems support different configurations depending on the software running on them, which makes vulnerability discussions dependent on specific use-cases and not general.

[6]Current processors can support up to 52 bits.
[7]Recently, Intel announced a 5-level page walk, but currently, no processors support this feature.

Table 1. Page Walk Configurations in a Modern Platform

EFER.LMA	CR4.PAE	CR4.PSE	PDPE.PS (PDPE1GB)	PDE.PS	TYPE	Page Size	Linear Addr Length	Phys. Addr Length	Mode
0	0	0	X	X	0	4 K	32-bit	32-bit	A
0	0	1	X	0	1	4 K	32-bit	32-bit	B
0	0	1	X	1	2	4 M	32-bit	40-bit	B
0	1	X	X	0	3	4 K	32-bit	36–52 -bit	C
0	1	X	X	1	4	2 M	32-bit	36–52-bit	C
1	1	X	0	0	5	4 K	48-bit	36–52-bit	L
1	1	X	0	1	6	2 M	48-bit	36–52-bit	L
1	1	X	1	X	7	1 G	48-bit	36–52-bit	L

3.5 Transactional Synchronization Extensions

Intel ***Transactional Synchronization Extensions*** (**TSX**) is an extension to the x86-64 **instruction set architecture** (**ISA**) that simplifies thread synchronization by adding a capability to identify execution areas that must be serialized, known as *lock elision* [19]. A transactional memory provides hardware-guaranteed atomicity to simplify concurrent programming that accesses shared data. Thus, a hardware mechanism must detect access (read/write) conflicts and undo any changes made to shared data. To implement this, major changes are required in the microarchitecture including caches. For instance, speculative values in a transaction must be buffered, and therefore remain not visible to other active threads until the values are final and can be committed.

Intel TSX provides the following two interfaces:

(1) **Hardware Lock Elision** (**HLE**)—an interface based on the instruction prefix that is backward compatible with processors that do not support TSX; and
(2) **Restricted Transactional Memory** (**RTM**)—a new instruction set interface that provides greater flexibility in implementing transactional memory.

The HLE interface simply holds a write to a lock allowing multiple threads to enter a critical region, and only restarting those transactions that fail. The RTM interface is more elaborate providing the option for the software to execute a fallback code in case of a transaction failure. Given that a transaction failure triggers a fallback code that can bypass exception handlers in the operating system, this seemingly simple caching of the rollback state needed to support the RTM interface created another opportunity to perform the side-channel attack of probing the OS page walk without triggering OS-visible faults. This can be abused to bypass a security mechanism known as ***Kernel Address Space Layout Randomization*** (**KASLR**), which will be discussed in Section 4.5.[8]

3.6 Branch Target Buffers

Branch mispredictions cause severe degradation in performance in a modern highly pipelined processor. An unconditional branch[9] interrupts the current instruction fetch operation and restarts the instruction window from a new memory location. Furthermore, any instructions that have been fetched beyond the unconditional branch must be discarded. A conditional branch that is taken can be even more disruptive as it must often wait for operands to be generated or status bits to be set before the target address of the branch can be determined. This means that the

[8]Before KASLR, TSX (see Section 3.5) was already proven to be an excellent mechanism for exploiting side-channel vulnerabilities [35].
[9]An unconditional branch "directly" specifies a target address without having to resolve an address.

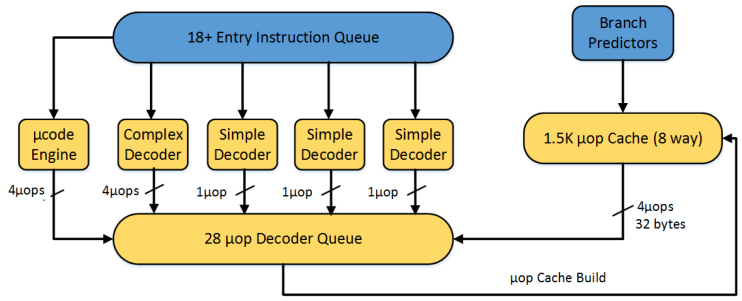


Fig. 4. Instruction decoding pipeline for the sandy bridge.

processor may have already fetched and partially executed many instructions beyond the branch. Thus, considerable effort has been devoted to reducing the performance cost of branches. Modern processors include **Branch Prediction Unit** (**BPU**) as a means to improve performance. The BPU has two main tasks. First, before the instruction stream is decoded, a prediction is made as to whether there are any branches coming up. For unconditional branches, a prediction is made on the branch target and then the next instruction is fetched from that location. For conditional branches, a prediction is made as to whether or not the branch will be taken. Although the inner workings of BPUs are highly proprietary, these units use an internal cache known as ***Branch Target Buffer*** (**BTB**). The BTB is used to cache addresses of recently executed branch instructions and their target addresses. This allows the fetching of the target instructions to start immediately.[10, 11]

The BTB is shared by the threads in a core, and is usually not flushed when a context switch occurs between applications running in that core. This means that information leakage from one application to another through a BTB timing side-channel is possible, as demonstrated in [1, 2]. Covert-channels using the BTB characteristics were also discussed in [26, 28]. Section 4.8 elaborates on the abuses of BTB.

3.7 Instruction Decoding Cache (Microcode Cache) and Small Loop Cache

The CPU fetches x86 instructions from memory and decodes them into micro-operations (μops) that are executed. This is achieved by multiple decoders that read the x86 instructions and generate regular, fixed length μops that are natively processed by the underlying hardware.

Figure 4 shows the instruction decoding pipeline for Sandy Bridge, which is the first microarchitecture to include a microcode (or μop) cache. For a complex instruction, the *μcode Engine* generates up to 4 μops/cycle, while a simple instruction generates one μop. The *μop Cache* stores instructions in their decoded form (similar to a trace cache or a basic block cache, explained in Section 3.8), and it is complementary to instruction caches.[12] All the μops from the front-end (i.e., decoding pipeline) and the μop Cache are ultimately delivered to the *μop Decoder Queue* as shown in Figure 4. The μop Decoder Queue also acts as a cache for small loops, as done in Nehalem.

[10]In the BTB structure, the number of entries and the number of bits are used for the address field matching and may vary by microarchitecture. Most modern microarchitectures use some set-associativity mapping for the BTB. Moreover, the BTB does not store all the address bits in the tag to save space.

[11]The BTB and the Branch Predictor could be two distinct units implemented as a small cache and an FSM, respectively, instead of a single Branch Predictor Unit. BTBs can also exist irrespectively of the presence of a branch predictor (e.g., just for jump instructions).

[12]An interesting point about the μop Cache is that it does not require other complex changes (such as having a trace BTB). The microarchitectures prior to Sandy Bridge had simpler forms of the μop Cache (e.g., Meron had Instruction Loop Buffer and Nehalem had a small loop cache).

Since an instruction stream is divided into multiple fixed numbers of bytes of instructions to be decoded, referred to as the *instruction window,* the mapping between the instruction cache and the μop Cache occurs at the granularity of an instruction window (e.g., Sandy Bridge has an instruction window of 32-byte instructions). After an entire instruction window is decoded and sent to the back-end for execution, it is inserted in the μop Cache as shown in Figure 4. This insertion is performed in parallel with the execution pipe, and therefore does not impose any extra delays. Programs that use instructions that average over 4 bytes (i.e., modern AVX instructions) benefit the most. There are many other benefits of the μcode Cache, but they are beyond the scope of this survey.

The μop Cache entries are addressed by the **Instruction Pointer** (**IP**) of the first decoded instruction from the ISA stored in the entry. Each entry also includes the number of valid μops and the length of the ISA instructions decoded. Note that there are no partial hits, meaning the full instruction window is either cached or not cached. Moreover, privilege level transitions do not cause flushes in the μop Cache, and cache evictions to occur based on the size of the instruction window. However, context switches (e.g., `mov CR3`) typically caused the μop Cache to be flushed.[13]

The μop Cache is co-located with the L1 instruction cache, and is organized as sets and ways with a limited number of μops per entry. This means that an instruction window can span multiple lines in the cache and a hit may take more than one cycle. There is also a maximum number of μops the cache can provide for each instruction window. If a given instruction window requires more than the maximum, it cannot fit in the μops Cache and will be decoded normally. Section 4.6 will discuss a security issue related to different semantics cached decoded instructions can have, which have a different behavior based on the privilege level.

3.8 Trace Cache

A *Trace Cache* differs from a conventional instruction cache in two basic ways:

(1) It stores μops instead of x86 instructions defined by the ISA; and
(2) It organizes code based on the expected execution flow rather than by memory locations.

One of the critical differences between the Trace Cache[14] and the μop Cache is that the latter is meant to augment a traditional front-end while the former was meant to replace it. In a conventional processor, the critical path for performance includes branch prediction (see Section 3.6), fetching of raw program bytes from the instruction cache, and decoding of x86 instructions to generate a stream of μops for OoO execution in the back-end. The Trace Cache improves the decoding of x86 instructions by caching the already decoded results as groups of logical μops that are stored together to represent a trace segment. This allows a μops trace segment to be fetched from the Trace Cache rather than re-fetching and re-decoding x86 instructions from the instruction cache.

In terms of security analysis, the Trace Cache is very similar to the μop Cache, and thus presents similar security concerns (see Section 4.6).

3.9 Latches, Queues, and Other Structures

A modern computer has many different structures that hold data, and the intention of this survey (or even possible) is not to cover all of them. However, the important aspect is that there are potential problems if stale data is held by a structure or repeated with latches. The following are

[13] An exception is when the PCID feature is used and there is no overflow in the number of stored IDs.
[14] Trace Cache is no longer present in modern Intel microarchitectures.

some examples of unconventional data structures that had security vulnerabilities associated with them:

— *Internal buses* that might hold previously transmitted values (i.e., stale data).[15, 16]
— *Latches* (also called repeaters) are used to optimize access paths when the data might not yet be ready. A latch will speculatively "repeat" the previously accessed data; therefore, optimizing some common computations without negatively affecting the performance of other computations. The problem with repeaters is that the speculatively "repeated" data might be considered as a secret to the currently executing computation. The MDS class of issues uncovered a few vulnerabilities related to latches [10];
— *Queues* that either leverage internal buffers to store data or delay actions based on internally controlled values, and therefore, might trigger actions after the data changed on the system. These are also an unexpected vectors and are discussed in Section 4.12;
— *Prefetching* of values that may cause side-effects that are undesirable, as discussed in Section 4.7.

It is very hard to test all the possible paths when reverse engineering a given hardware implementation. Therefore, many of these issues have remained hidden for years. In addition, fixes provided by different companies might not be comprehensive, as the recent speculative execution attacks demonstrated (see Section 4.9), since they might address only specific paths (versus systematically eliminating the entire vector).

4 SURVEY OF CACHE-RELATED SECURITY WEAKNESSES

This section discusses various weaknesses exposed by a platform due to the presence of different caches. Such weaknesses become security vulnerabilities in certain software contexts depending on the system configuration, e.g., a cache split is only a problem if a security property of the software depends on maintaining coherency.

Table 2 summarizes the weaknesses discussed in this section and their type of cache attacks as categorized based on the security vulnerability taxonomy discussed in Section 2.2.

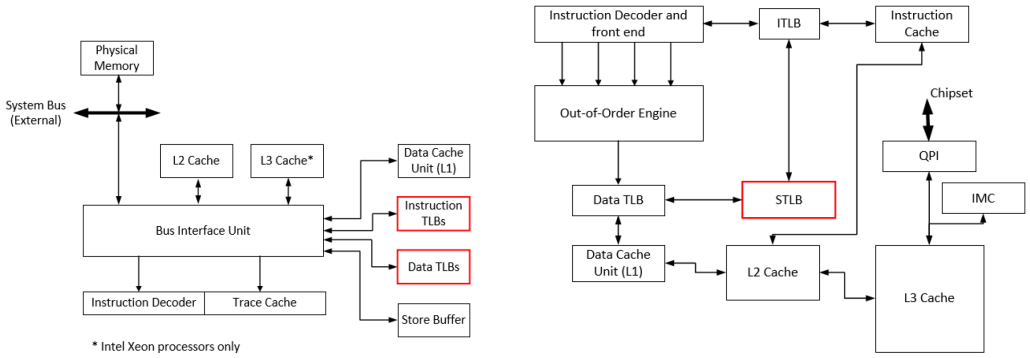
4.1 TLB De-synchronization

TLB de-synchronization is a Type 1 cache attack that was introduced for the first time in 2005 and named *Shadow Walker* [66]. In a Type 1 cache attack, the attacker's objective is to force the system to have different copies of the same data in different caches (i.e., create an async). This means different consumers of such a data will see different values. Therefore, a consumer that tries to verify data integrity or maliciousness, such as an anti-virus software, will get a different copy of the data than the ones actually used by the other parts of the system thereby bypassing the protection mechanism. In order to understand how TLB de-synchronization can be used as an attack, Figure 5(a) shows the cache organization for early x86 architectures[17] that have split TLBs consisting of I-TLB and D-TLB. Normally, the loader first loads the program causing its page-table translations to be cached in the D-TLB. Afterward, page table translations are cached in the I-TLB as the program executes. As a result, the corresponding entries in the I-TLB and D-TLB will be identical and thus synchronized as shown in Figure 6(a).

[15]Even when the hardware designer specifically "zeroes" (i.e., erase) the bus content, in practice the manufactured CPU might not have the logic in its final form due to manufacturing process optimizations. This kind of optimization is similar to a compiler removing code that it deems unnecessary. Such a case of bad hardware manufacturing optimization led to one of the vulnerabilities associated with the class named **Microarchitectural Data Sampling** (**MDS**) [10].
[16]The specific case involves AES-NI and FPU instructions that shared a common bus.
[17]Up to the Nehalem microarchitecture introduced in 2008.

Table 2. Categorization of Weaknesses and Their Associated Types of Attacks

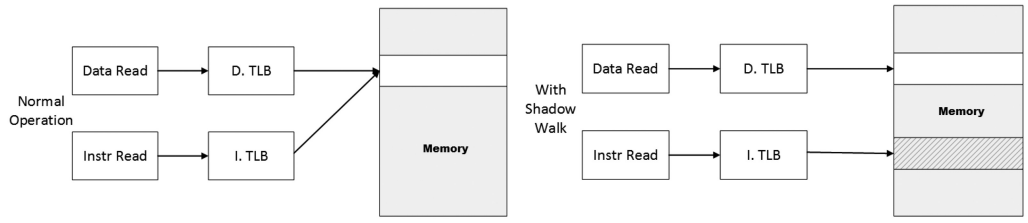
Attack Type	Key Characteristics	Example Weaknesses	Sub-Section	Ref
Type 1	Creates an async between I-TLB and D-TLB to hide malicious code	TLB De-synchronization	4.1	[66]
	Creates an async between caches and the system memory to hide malicious code	Cache-System Memory Async	4.2	[6]
Type 2	Uses software accessible side-channels in different caches to discover secrets from other applications in the system	Cache-based Side-Channels	4.4	[58]
	Uses transactional memory capabilities as a side-channel mechanism	Transactional Memory	4.5	[45]
	Leverages the branch prediction as a side-channel mechanism	BTB-related	4.8	[27]
	Leverages the page walk hardware as a side-channel mechanism	Memory Management Unit	4.10	[34]
	Leverages the Integrated Graphics Processing Unit to enhance side-channel attacks	Integrated Graphics Processing Unit (GPU)	4.11	[31]
	Leverages the Hardware Store Elimination Feature as a side-channel	Hardware Store Elimination feature	4.13	[25]
Type 3	Abuses branch prediction and speculative execution creating a covert-channel between the microarchitecture and architecture views of the system to leak secret data. Oftentimes requires Out-of-Order (OoO) execution.	Cache-based Covert-Channel Attack via Speculative Execution (also known as transient execution attacks)	4.9	[39]
Type 4	Uses memory access prioritization (based on cached results) to deny (or delay) access to the system memory	Memory Performance Attacks	4.12	[56]
Type 5	Leverages a cache hit in a different CPU mode to overwrite protected memory	Cache Poisoning	4.3	[10]
	Leverages the context difference of an instruction cached in user mode and later executed in kernel mode	Security Issue of μop Cache	4.6	[20]
	Leverages the prefetch mechanism to find mapped (valid) memory areas in the kernel	Prefetch-based Side-Channel	4.7	[37]



(a) Pre-Nehalem TLB Organization. (b) Post-Nehalem (2008) TLB organization.

Fig. 5. TLB Organizations.

However, separate TLBs creates the possibility of having an asynchrony between them, where an access to the same linear address is translated into different physical addresses depending on the access type (data or instruction). This problem is illustrated in Figure 6(a), where I-TLB and D-TLB contain different physical addresses for the same linear address. This is a security issue because it permits an attacker code to be hidden from a security software. For example, an anti-virus software reads memory to search for a malicious code, which involves accessing data using D-TLB, while



(a) A normal operation where TLBs are synchronized.

(b) A computer infected with Shadow Walker, where TLBs are not synchronized

Fig. 6. TLB sync versus async.

code execution involves fetching instructions using I-TLB. If there is a TLB de-synchronization, instruction fetch and data access will occur from different physical addresses even though they have the same linear address.

Shadow Walker forces a TLB de-synchronization and protects both its code execution and data fetches by performing the following operations:

(1) Trap on a page or a protection fault caused by access to the hidden memory—This is performed by marking the hidden memory as inaccessible using the paging protection bits and hooking the fault handler in the OS;
(2) Determine whether the trap was caused by a data access or an instruction fetch—This is done by checking if the *PC* (EIP/RIP) register matches with the *CR2* register that contains the ***Page Fault Linear Address*** (**PFLA**) (i.e., the address that generated the fault). If the addresses are the same, the trap was caused by an instruction fetch; otherwise, it was caused by a data access;
(3) Prime the appropriate TLB—For a data access, the corresponding page table translation for the address is cached in the D-TLB by setting, reading, and then clearing its data access bits. For an instruction fetch, the corresponding page table translation for the address is cached in the I-TLB by executing a single instruction using the **Trap Flag** (**TF**), which enables single-stepping; and
(4) Repeat for all addresses of interest.

As discussed in Section 3.3, the possibility of TLB de-synchronization was eliminated when S-TLB was added to the Nehalem microarchitecture in 2008 as shown in Figure 5(b).[18] However, Torrey demonstrated at Black Hat Las Vegas in 2014 that forcing a TLB de-synchronization was still possible due to ***Extended Page Tables*** (**EPT**) introduced to support virtualization [68]. As discussed in Section 3.4, when VT-x is enabled, the address translation process changes slightly due to the existence of the EPT because a **Virtual Machine ID** (**VMID**) tag is also used during the translation lookup. This avoids TLB flushes that would normally occur without VMID and improves performance.[19] The behavior of the S-TLB also changes due to this new layer. Since the EPT provides more granularity than traditional paging, I-TLB and D-TLB entries that have conflicting security permissions will not be merged and stored in the S-TLB unless their VMID tags also match. In essence, this means a hypervisor is still able to create a split because the TLB entries will not be merged and can be prevented from being flushed, even in modern platforms.

[18]However, older systems are still susceptible to this problem.
[19]Address space changes cause TLB flushes of pages not marked as global to avoid the address translation of one process affecting another process.

A TLB de-synchronization requires a high-level of privilege, which means that the system has already been compromised and thus an attacker has a lot of control. Therefore, TLB de-synchronization is not considered as a vulnerability. Instead, it is an example of unexpected, or at least undocumented, behavior that security software should be aware of. As discussed in the Online Supplement, such an unexpected behavior can also be leveraged to implement security mechanisms.

There are other attacks that leverage the different behaviors of TLBs. For example, Hund et al. demonstrated a KASLR bypass by leveraging timing issues in accessing pages cached in the TLB [44]. There are more recent ones, such as *TLBleed* released in 2018 [33].

4.2 Cache-system Memory Async

In the Frozen Cache research [59], a cache/memory async was created to keep the encryption keys only in the cache and not in DRAM. This created an opportunity for attackers to perform a Type 1 cache attack by hiding any data from physical memory acquisition. The Frozen Cache research suggested how it is possible to create such an async, but it does not prove it really happened.

Branco and Barbosa proved the possibility of creating such an async between the cache and the system memory [6]. Their solution to verify an async between cache and system memory was based on the page walk mechanism explained in Section 3.4 [6]. However, some additional explanation is necessary to better understand their findings. Their research demonstrated that exiting the no-fill mode causes a write-back to synchronize the cache contents that were changed (see Section 3.1) [6]. They also showed that the `invd` instruction does not invalidate the cache if executed during the no-fill mode. These two undocumented behaviors discovered were used together with the next steps for the final proof. First, an assumption was made on the behavior of the hardware as well as proving such an assumption was correct. The assumption was that the page walk hardware can be forced to not use the cache. The rest of the explanation was then based on a system with the PAE enabled (see Section 3.4). This affects the translation mechanism, but has no effect on the cache. If it is possible to prove that the page walk is not using the cache, then it proves that an async exists between the cache and the system memory. The rationale for this proof is discussed next.

Table 1 shows the different configuration options for the page walk process in a platform. In a 32-bit system (when EFER.LMA = 0 and CR4.PAE = 1), the 3-level page walk is slightly different than the ones shown in Figure 3: The CR3 points to the PDP Table instead of the PML4 Table, which contains the translation information shown in Figure 7. The proof is based on the fact that **Page Directory Pointer Table Entry** (**PDPTE**) 0 is the first structure referenced in a page walk and a fault will occur if it is marked as non-present. This structure is defined in the architecture manuals as shown in Table 3.

When the **Present** (**P**) bit of a PDPTE is cleared, it means that this structure is no longer valid and any references to it will generate a **page fault** (**#PF**). Thus, it is possible to demonstrate that there is an async between the cache and the system memory by performing the following steps:

(1) Disable interrupts and run only a single thread. This prevents anything else in the system from interfering with the test.
(2) Mark the PDPTEs as write-back. This means that when the PDPTEs are accessed, they will be brought into the cache, and when they are modified they will be updated in the cache but not in the memory.
(3) Using the `wbinvd` instruction,[20] write back all the dirty cache blocks and invalidate the cache so that it is empty.

[20]Note that `wbinvd` returns immediately rather than after the invalidation completes [13], so a delay is necessary.

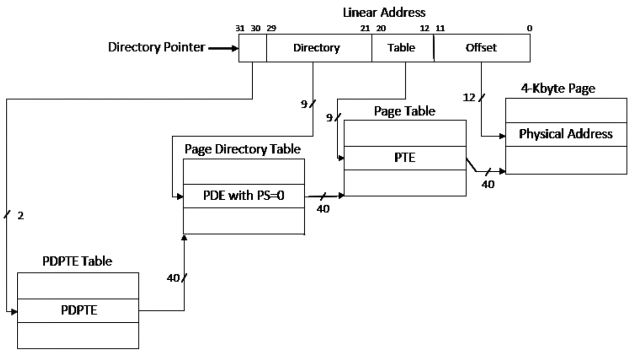


Fig. 7. PAE translation.

Table 3. Format of a PAE PDPTE

Bit	Name	Description
0	P	Present – must be 1 to reference a page directory
2:1	Reserved	Must be 0
3	PWT	Page-level write-through – indirectly determines the memory type used to access the page directory referenced by this entry
4	PCD	Page-level cache disable – indirectly determines the memory type used to access the page directory referenced by this entry
8:5	Reserved	Must be 0
11:9	Ignored	Could be any value, since it is ignored
(M-1):12	Address	Physical address of 4-KByte aligned page directory referenced by this entry (M is at most 52)
63:M	Reserved	Must be 0

(4) Access the PDPTEs forcing them into the cache. The proof of the async is based on how the system will behave once the PDPTEs are modified.
(5) Clear the P-bit in the PDPTE. Since the entry is cached as a write-back, any updates will occur only in the cache.
(6) Perform random memory accesses using virtual memory to force a page walk but not enough accesses to fill the cache; otherwise, evictions would occur and the memory would be updated. No memory translations should occur if the PDPTEs from the cache has the same value as the ones from the memory and the system should reboot.

Since the above-mentioned steps did not cause the system to reboot, it means that the page walk mechanism did not use the cache. If the cache was used, no memory translation would have been performed and this would have generated a fault. Moreover, since no handler exists (because no page walk is possible), a double fault would occur and then a reboot. A test was also performed to force a write-back after clearing the P-bit to trigger such a reboot (in order to confirm the assumption that the reboot should happen). This demonstration showed that it is possible to force an asymmetry between the cache and the system memory.

An async can be used as an attack like the TLB de-synchronization attack discussed in Section 4.1.

4.3 Protected Memory Attacks using Cache Poisoning

Transitions to different modes of execution occur in architecturally defined ways. For example, as shown in Figure 8, a system running in the Protected Mode transitions to the **System**

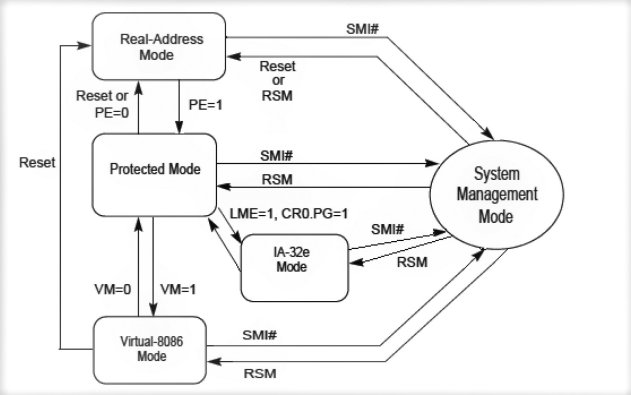


Fig. 8. Modes of execution and how they are related to each other.

Management Mode (**SMM**) when a **System Management Interrupt** (**SMI#**) is triggered. Afterward, executing the `rsm` (Resume from SMM) instruction will cause the system to transition back to the previously running mode. Given the different privilege levels of different modes (e.g., accessing the memory area exclusive to the SMM, known as System Management RAM, SMRAM, is not allowed in the Protected Mode), how caches behave during such transitions may create problems. An example of such a problem was identified in [7] and later proven to be a security issue in [78]. What the authors found was that cache entry created outside of SMM were not invalidated before entering SMM. Those entries were instead written back to memory *while* in SMM thereby overwriting the protected memory. Cache entries created by a less privileged mode are referred to as poisoned because they should not be trusted and used. This is a Type 5 attack where these memory writes can be abused to elevate the privileges of the attacker.

4.4 Cache-based Side-channels Primitives

Cache-based side-channel attacks discussed in this section refer to attacks that use cache-related operations and timing measurements to obtain information about the memory access history of a victim, which can be a process or a routine running in a different security context from the attacker. These Type 2 cache attacks utilize knowledge about memory accesses to leak the private data of a victim, such as secrets (i.e., a password) and other security assets (e.g., encryption keys).

Most cache-based side-channel attacks consist of three steps: (1) The attacker performs operations to create an initial cache state; (2) the victim is allowed to access memory either synchronously or asynchronously to cause cache loading/eviction; and (3) the attacker directly executes or indirectly triggers a probing routine that measures the timing of the cache operations to detect the change from the initial cache state and identify the memory access pattern of the victim's execution in Step (2).

There are three main approaches to cache-based side-channel attacks: Prime+Probe, Flush+Reload, and Flush+Flush. All three approaches belong to the same family, but differ in terms of attack condition, resolution, and stealthiness. There also exists the Evict+Reload option that requires a shared memory, usually a shared library or shared pages between **Virtual Memories** (**VMs**) as the measurement point. The attacker first evicts the shared memory from the cache set. If the victim accesses the shared memory, this will overwrite the attacker's data in the cache. The attacker can differentiate whether or not the victim accessed the shared memory by measuring

the time to access it. Evict+Reload is a newer option that is less dependent on specific instructions to flush the cache.

The *Prime+Probe* approach takes advantage of the memory-to-cache mapping correlation in direct-mapped or set-associative mapping, where a memory block can only be mapped to a specific cache set. Therefore, a cache with S sets and block size C, a block with address A can be evicted by the filling of another block with address B that is congruent with A mod $S \times C$. In a Prime+Probe attack, the attacker first fills the cache with blocks from a known starting address A in the main memory, referred to as the Prime step. Then, the attacker waits for a certain time window to allow for the execution of the victim. In the Probe step, the attacker accesses address A again and measures the access time to determine if the block remains cached or has been evicted. The latter case indicates that there is a high probability the victim has accessed the memory address that is congruent with A mod $S \times C$. The advantage of Prime+Probe is that it does not require shared memory between the attacker and the victim. However, the attack resolution is limited by the number of cache sets in the system and is less deterministic compared with the other two approaches.

Osvik et al. demonstrated that Prime+Probe can be used to attack an AES cipher based on statistical analysis and knowledge of the victim's memory-access pattern [58]. Given the fact that performance-oriented AES implementations typically use pre-computed lookup tables in memory to carry out encryption operations, such as ShiftRows, MixColumns, and SubByte, data-dependencies in memory access patterns can be used by cryptanalysis to reverse the cipher after the memory access information is obtained by the Prime+Probe side-channel attack.

Flush+Reload is a simpler and more deterministic approach than Prime+Probe. A Flush+Reload attack consists of three steps: (1) the attacker evicts a certain block or blocks from the entire cache hierarchy; (2) the attacker waits for a certain time window to allow for the execution of the victim; and (3) the attacker accesses the evicted memory addresses and measures the time to access each block to determine if it was cached, in which case it indicates the victim has accessed that specific memory location during the probe window in Step (2). Note that this approach requires memory blocks between the attacker and the victim to be shared, which is common in modern multi-user systems. Examples include shared modules between processes and page-sharing between virtualized guests, such as the **Kernel Same-page Merging** (**KSM**) feature that is widely used by the host to reduce the overall system memory footprint.

Flush+Reload can be used as a side-channel attack by itself as demonstrated in numerous published research efforts. For example, it was used to attack RSA in a certain versions of **GNU Privacy Guard** (**GnuPG**) with CRT-RSA optimization to extract private keys based on its data-dependent memory access encryption algorithm [82]. It was also widely used as a measurement approach in speculative covert-channel attacks (e.g., Spectre and Meltdown [30, 38, 39, 48, 51]) by deliberately triggering a data-dependent cache loading gadget since cache block filling occurs during speculative execution.

Flush+Flush is a close variant to the Flush+Reload approach except that in the last step the evicted block is flushed again instead of reloading. This approach relies on the fact that it takes a longer time to perform a flush operation (e.g., using `clflush`) when the cache block it tries to flush is actually in the cache. Therefore, this approach can also determine if the cache block that was originally evicted has been brought back in by the victim during the probe window. The major performance difference between these two approaches is that a Flush+Flush attack does not generate a large number of **last-level cache** (**LLC**) misses compared to a Flush+Reload attack, which can be detected using hardware event monitoring, e.g., using the performance monitor feature in Intel x86 processors [29]. Therefore, the advantage of Flush+Flush over Flush+Reload is attack stealthiness.

4.5 Transactional Memory Attacks

Transactional memory attacks are Type 2 cache attacks that exploit a side-channel behavior (e.g., timing) of memory address translation to extract valuable information, mostly regarding memory layout.

Jang et al. demonstrated that a side-channel attack on TLB-related address translations can break KASLR by leveraging Intel's TSX technology (see Section 3.5) [45]. The bases of this attack are

- Access a memory area that is currently mapped to the kernel address space (i.e., have valid translations) so that the address translations are cached in the TLB. Subsequent accesses to the same page will hit in the TLB, while accesses to unmapped areas will not affect the TLB;
- The contents of the TLB only depend on the completion of address translations, i.e., page walk, rather than the completion of accesses. For example, accessing privileged memory addresses in user mode will populate the TLB regardless of whether or not a page fault is generated.

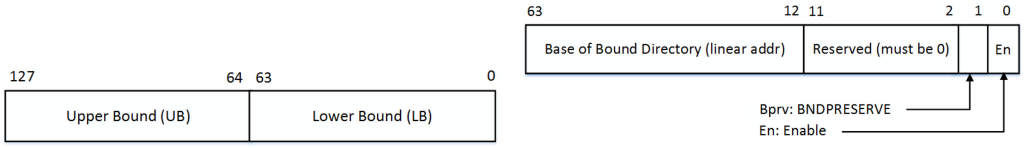
Based on the above, it is possible to probe a kernel address in user mode to determine whether an address is mapped or not based on the fact that less time is taken to generate page faults for mapped addresses than for unmapped addresses. However, the challenge of this attack is the small signal-to-noise ratio since a TLB miss only adds about 40 CPU cycles to the fault generation and handling process. If the memory access probe is timed by a user mode fault handler, there is less than a 1% difference in total time between trying to access mapped versus unmapped addresses due to the overhead required by the OS to dispatch the fault to a user mode handler.

This challenge can be overcome by taking advantage of the TSX feature of Intel processors. As explained in Section 3.5, TSX provides a hardware transactional memory support that allows the software to execute a fallback code in case of a transaction failure that bypasses exception handling of the OS. Therefore, when there is an exception during a TSX transaction, the execution is aborted and the user mode abort handler is directly invoked without the intervention of the OS.

The example below shows a simple code snippet that uses TSX `_xbegin()` and `_xend()` to perform a time measurement of a memory access probe that generates a fault (e.g., due to a privilege error when a user mode process tries to read a kernel mode address):

```
// Timer starts, rdtsc() returns the timestamp
uint64_t start = rdtsc();
// initiate TSX region
if (_xbegin() == _XBEGIN_STARTED) {
    // func() tries to access address probe_addr,
    // which is supposed to generate a page fault
    func(probe_addr);
    // commit TSX, which will not happen when exception is raised
    _xend();
} else {
    // TSX aborted; end timer and get the time duration
    uint64_t access_time = rdtsc_end() - start;
}
```

With TSX, the fault handling overhead is reduced to around 200 cycles, which results in more than 15% timing difference between accesses to mapped and unmapped addresses. In this case, the kernel memory layout can be ascertained very quickly and accurately to break KASLR.



(a) Format of MPX Bounds Register. (b) Format of MPX Bounds Configuration Register.

Fig. 9. MPX registers.

4.6 Security Issue of the Microcode Cache

As discussed in Section 3.7, the μop Cache enhances the front-end performance by not having to re-decode instructions that have already been executed. However, the μop Cache is not flushed upon context switches; therefore, instructions with context-dependent decoding introduce potential security issues. A good example of such an issue involves the ***Memory Protection Extensions*** (**MPX**) technology of 6th generation Intel processor family [20]. MPX is a new security feature introduced to the Skylake microarchitecture to protect against memory corruption attacks that exploit buffer overflow/underflow by checking whether runtime memory references are within the desired boundaries.

The hardware-assisted MPX introduces four new 128-bit *bounds registers* (BND0–BND3), where each register contains a 64-bit **Lower Bound** (**LB**) and 64-bit **Upper Bound** (**UB**) to store the boundaries of a memory buffer. This is shown in Figure 9(a). New instructions are also added to support MPX operations, such as setting the bounds (`BNDMK`), checking lower/upper bounds (`BNDCL`/`BNDCU`), load/store bounds from/to memory (`BNDMOV`), and so on. MPX also supports a two-level mapping that maps a pointer to a pointer address to its corresponding bound data structure in memory.

The particular security issue related to MPX involves the initialization behavior of the bounds registers when branch instructions are executed. Since these registers hold the boundaries of buffers used by the program, which can be automatically defined by the compiler, and given buffer addresses are checked to be within bounds before any accesses can be made. The bounds registers are considered initialized when LB is set to 0 and UB is set to all ones, which represent access to the entire address space. Based on this, the initialization behavior of branch instructions (such as `CALL`, `RET`, `JMP`, and `Jcc`) is determined by both the `BND` prefix (F2h) used together with the specific instruction (e.g., `BND RET`) and the BND configuration register as follows:

— If the BND prefix is present, the bounds registers remain unchanged when a branch instruction executes;
— Otherwise, the bounds registers will be initialized upon execution of a branch instruction when the *BNDPRESERVE* bit of the BND configuration register is 0, and will remain unchanged when the *BNDPRESERVE* bit is 1 (see Figure 9(b)).

Intel added this specific behavior based on the branch instruction in order to provide compatibility with legacy libraries that were not recompiled to use MPX. This allows a new code to deactivate MPX while branching to a legacy library that does not support it.

There are two different BND configuration registers: one for user mode and one for kernel mode. For user mode, the configuration register is defined as *BNDCFGU*, and for kernel mode, it is implemented as an MSR named *IA32_BNDCFGS*. These registers have the same layout for both modes as shown in Figure 9(b). Therefore, a branch instruction can be executed with different behaviors depending on how the bounds registers are initialized for different privilege levels.

A consequence of the context-dependent decoding for MPX is that a user mode non-prefixed branch instruction (i.e., *BNDPRESERVE* is 0) might initialize the bounds register while in the kernel mode (i.e., *BNDPRESERVE* is 1) the same instruction may not. Moreover, if the same branch instruction is first executed in the user mode and then executed in the kernel mode,[21] the kernel mode execution will hit in the μop Cache and reuse the decoded μops from the user mode execution. Therefore, the bounds registers will be initialized when it is not supposed to, which is a security issue because the initialization of bounds registers allows access to the entire memory range bypassing the protection of MPX.

4.7 Prefetch-based Side-channel Attack Against KASLR

Prefetching is supported across different architectures (e.g., x86, ARM). Hardware prefetching is carried out transparently by the CPU, while software prefetching is performed using the `prefetch` instruction that loads a certain block of data from memory to cache. This instruction can further specify which level of cache to prefetch into. For example, current Intel CPUs provide multiple prefetch instructions—`prefetcht0` (all cache levels), `prefetcht1` (level 2 cache and higher), `prefetcht2` (level 3 cache and higher), and `prefetchnta` (non-temporal cache structure, which is a hint to the processor that data will be only used once).

There are two properties of prefetching that make it a good candidate to be used in side-channel attacks:

— Prefetching instructions merely provide hints and can be ignored by the processor,[22] and therefore do not have a deterministic behavior. They also do not affect program behavior according to Intel's developer manual, and therefore do not generate faults regardless of the validity and privilege levels of the addresses to be prefetched.
— The execution time of a prefetch instruction depends on the actual fetched flow (i.e., address translation, data movement) and can be accurately measured.

Given these attributes, Gruss et al. demonstrated that a Type 5 attack is possible through a prefetch side-channel to break KASLR on platforms running Linux OS with Intel CPUs [37]. The attack is carried out by a user mode application that uses `prefetch` to probe the kernel memory range and retrieves the virtual address of the targeted kernel driver, which is not supposed to be known by user processes due to KASLR.

The attack involves two stages to exploit the side-channel behavior of `prefetch`. In the first stage, a code sequence shown below is used to measure the time taken to prefetch an arbitrary virtual address that belongs to the kernel memory.

```
1    ; rcx = kernel address
2    mfence
3    rdtscp
4    mov ebx, eax
5    cpuid
6    prefetchnta [rcx]
7    cpuid
8    rdtscp
9    mfence
10   sub eax, ebx
```

[21] Because the microcode cache lookup uses the IP, this bug would only happen if both the kernel and user modes were sharing addresses for the instruction.
[22] This is to accommodate future optimizations that would otherwise be impossible with guaranteed execution of prefetch instructions.

The code uses `rdtscp` to measure the time taken to execute the `prefetchnta` instruction. Note that `mfence` is used to serialize all the memory loads and stores to increase the measurement accuracy, while `cpuid` is used to serialize the prefetching itself since it is not serialized by other memory fences.

The timing result from such a measurement reflects the address translation behavior of the prefetch flow, i.e., the target virtual address of `prefetchchnta` needs to be translated into the physical address to locate the memory block. This will either hit in the TLB or go through the multi-level page translation. When the user mode code prefetches from an arbitrary kernel address space, one of the following three possibilities occurs: (i) complete the translation by hitting in the TLB, (ii) go to the last level of the paging structure in case of a mapped address, at which point the privilege level mismatch is known, or (iii) stop at the level with an invalid paging entry in case of an unmapped address. In all cases, no actual cache loading will take place. However, based on the address translation flow, the timing results are statistically distinguishable between the cases of mapped and unmapped kernel addresses.

Since kernel drivers (modules) are loaded sequentially, the published attack by Gruss et al. uses 2 MB steps to search through the designated kernel address range to locate the regions that are mapped to physical memory [37]. After narrowing down the address range in the first stage, the second stage of the attack further pinpoints the exact address of the targeted driver using the *Evict+Prefetch* approach, which involves the following three major steps:

(1) Evict all cached blocks that belong to the kernel as well as address translations cached in TLB by accessing a large buffer in user mode;
(2) perform a system call to the targeted driver; and
(3) measure the time required to prefetch a kernel address p within the mapped area obtained from the first stage.

Step 2 will cache both the driver code and the address translations of the driver regions. Therefore, if the kernel address p probed in Step 3 is within the target driver range, prefetching will take less time than the baseline case without the system call. Afterward, the lowest virtual address of the targeted driver can be determined using a fine-stepped search, which gives the exact base address of the driver effectively breaking the mitigation objective of *KASLR*.

4.8 BTB-related Attacks

As introduced in Section 3.6, BTB is a special cache that stores target addresses of recently executed branches. This allows a branch instruction to execute faster when its address hits compared to when it misses in the BTB. Since BTB is shared across different security contexts on the same core, whether a specific branch instruction has been executed in the victim can be determined by tailoring it to hit the same BTB entry and measuring its execution time. This is a Type 2 attack because knowing whether a given branch has been executed may reveal pieces of information from the target process. A simple example is when a branch is a conditional based on one bit of a secret key.

BTB entries are tagged with certain bits of branch addresses; therefore, an attacker controlled branch can cause a BTB collision with the targeted branch in the victim when these bits match. Figure 10 shows an example code snippet, where the attacker code executes right after the victim's function. In this example, whether the address of *attacker_jmp* is going to collide with the address of *victim_jmp* can be determined by measuring the execution time of the former. If the jump target offset *T2* in the attacker and the victim are the same, the attacker will execute faster than the baseline (which is measured before running the victim).

```
<victim_func>                    <attacker_func>
push rbp                         push rbp
mov rbp, rsp                     mov rbp, rsp
...                              rdtscp
...                              ...
<victim_jmp>:                    <attacker_jmp>:
jmp <victim_jmp + T2>            jmp <attacker_jmp + T2>
nop                              nop
...                              ...
nop                              nop
<victim_jmp + T2>:               <attacker_jmp + T2>:
 nop                              nop
...                              rdtscp
...                              ...
pop rbp                          pop rbp
ret                              ret
```

Fig. 10. Attacker and victim routines.

One such scenario of a side-channel attack is to break KASLR by pin-pointing the virtual address of a known branch in the victim's execution flow. For example, Evtyushkin et al. demonstrated that KASLR in modern 64-bit Linux OS can be completely bypassed, and user mode ASLR can be weakened by exploiting the BTB-related side-channel behavior [27].

The attack to break KASLR is carried out on a platform with an Intel Haswell processor running Linux kernel version 4.5. Besides the BTB side-channel mentioned above, the attack is also based on the fact that the Haswell platform uses only bits 0–30 of the virtual address for BTB entry tagging, while bits 31–47 are ignored. This allows a cross-mode BTB collision to occur between a user mode branch and a kernel mode branch. Meanwhile, KASLR in 64-bit Linux only randomizes bits 21–29 of the kernel virtual address, which makes it possible to recover the full virtual address of a specific kernel mode branch instruction in user mode.

The attack contains the following five steps:

(1) Find a target branch (a direct jump in this case) in the kernel code, where its execution can be triggered by a user mode application, e.g., through a system call;
(2) assuming the targeted kernel mode branch has virtual address K, the user mode attacker allocates memory to load a code block containing a direct jump at virtual address A and makes sure that (i) the bits 0–19 are the same between A and K; (ii) the jump instruction at address A jumps to a different offset than the jump at address K;
(3) the user mode attacker initiates the execution of the targeted kernel mode branch at address K by calling the identified system call;
(4) the user mode attacker runs the code block containing the jump at address A and measures its execution time. Repeat Steps 3 and 4 multiple times to obtain a statistically stable results; and
(5) the code block containing the jump at address A is moved by a 2 MB offset each time and the above test is repeated to go through all possible combinations for the bits randomized by KASLR (i.e., bits 21–29). Identify BTB collision cases that require a longer execution time for the jump instruction.

There are a total of 512 possible addresses for A and the attack can be carried out within a very short time (~60 ms).

Table 4. Vulnerability Identifications and Names

CVE	Intel Nomenclature	Other Known Names
CVE-2017-5753	Bounds Check Bypass	Variant 1/Spectre v1
CVE-2017-5715	Branch Target Injection	Variant 2/Spectre v2
CVE-2017-5754	Rogue Data Cache Load	Variant 3/Meltdown
CVE-2018-3639	Speculative Store Bypass	SSB
CVE-2018-3640	Rogue Register Read	-
CVE-2018-11091	Microarchitectural Data Sampling Uncacheable Memory	MDSUM/RIDL
CVE-2018-12126	Microarchitectural Store Buffer Data Sampling	MSBDS/Fallout
CVE-2018-12127	Microarchitectural Fill Buffer Data Sampling	MFBDS/RIDL
CVE-2018-12130	Microarchitectural Load Port Data Sampling	MLPDS/RIDL
CVE-2019-11135	TSX Asynchronous Abort	TAA/RIDL
CVE-2019-1125	SWAPGS	-
CVE-2020-0549	L1D Eviction Sampling	L1Des/RIDL
CVE-2020-0548	Vector Register Sampling	VRS/RIDL
CVE-2020-0550	Snoop-assisted L1 Data Sampling	Snoopy
CVE-2020-0543	Special Register Buffer Data Sampling Advisory	SRBDS/Cross-Talk
CVE-2020-12965	Transient Execution of Non-canonical Accesses	-

The same approach can be applied to attack user mode ASLR by running the attacker process on the same core as the victim process. The limitation of this approach is that ASLR in 64-bit Linux randomizes 28 bits of the virtual address (i.e., bits 12–39), while the lower 31 bits are used for BTB addressing making it possible to only recover part of the randomized bits. However, this is still a considerable reduction of the ASLR entropy making it computationally possible to brute force the remaining bits.

4.9 Cache-based Covert-channel Attack via Speculative Execution

Cache-based covert-channel attacks that exploit speculative execution are a fairly new class of attacks. These attacks are known as *Spectre* (with variants [38, 39, 48, 64]), *Meltdown* [39] (with variants [30, 38, 51]), and ***L1 Terminal Fault*** (**L1TF**) [17], also known as *Foreshadow*. Additional speculative attacks are still being discovered that leverage different caches and techniques [9, 22, 63, 71, 73], and these are all examples of Type 3 attacks. These attacks are also known as *transient execution attacks* because they exploit microarchitectural actions that are supposed to be discarded [8]. The reason why these attacks continue to be found is because they affect different caches in many different systems and implementations. They exploit prefetchers, BTBs, internal CPU buffers, and even latches. Instead of covering the specifics of each case, this subsection provides a general understanding on how this class of attacks is performed.

Although the focus of this discussion is on Spectre, Meltdown, and Foreshadow/L1TF, there are numerous other attacks based on cache-based covert channels. Table 4 shows some of the different vulnerabilities and the names that were given to them either by the researchers who first discussed the issues or by Intel). They are also known by **Common Vulnerabilities and Exposures** (**CVE**) IDs.[23]

Speculative execution is one of the most important techniques of modern processors to improve CPU performance across various architectures. It allows instructions to be executed microarchitectural in advance of knowing whether or not they will be architecturally committed. The CPU executes these instructions without any security checks because their results will be dropped if the speculation is found to be incorrect, and therefore will not influence the final computation. In principle, speculative execution does not cause security issues as long as its microarchitectural states are not exposed to the execution. However, one exception to this is that cache blocks are

[23]CVE is a list of publicly known cybersecurity vulnerabilities.

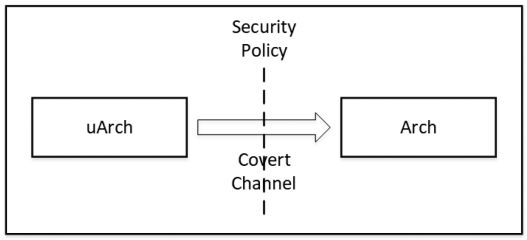


Fig. 11. Schematic of cache-based covert-channel.

loaded during speculative execution. This means that the uncommitted microarchitectural states, which should not be exposed, can leave measurable traces for architectural probing.

Note that the OoO execution model allows instructions to be dispatched to execution units in an order that is potentially different from a given program order. While it is possible to design a CPU that is OoO but not speculative or one that is speculative but in-order, the discussion in this section applies to CPUs that use both speculation and OoO. Some of the issues discussed, such as Meltdown are specifically dependent on the presence of OoO execution.

Besides the cache-based covert-channel, another key element of this type of attacks is to gain control over the speculative execution so that (1) the speculative cache loading is data-dependent and (2) the data is from a different security domain that should not be exposed to the attacker. This class of attacks often uses some of the classic cache-based side-channel techniques (e.g., Flush+Reload and Flush+Flush) as a measurement approach, but it is still essentially a covert-channel attack since its essence is to break the isolation between microarchitectural and architectural states (see Figure 11).[24]

Although both Spectre and Meltdown use similar cache-based covert-channels, their variants have different approaches for triggering the desired speculative execution. Spectre, with two variants, induces a controlled speculative execution in the victim by exploiting branch prediction, whereas Meltdown exploits speculative behavior in a potentially fault-triggering condition in the attacker itself.

Spectre

Both variants of Spectre leverage branch prediction, which is one major aspect of speculative execution. Spectre Variant I misuses the conditional branch predictor.[25] The code snippet shown below could be vulnerable for attack by Spectre Variant I due to its code structure.

```
//STRIDE is a constant defined as integer times of cache block size
//untrusted_index is untrusted input from attacker
if (untrusted_index < array1_size)
    data = array2[ array1[untrusted_index] * STRIDE ];
```

The conditional branch in the third line checks if `untrusted_index` is within the valid range. Since this is attacker controlled, the conditional branch can be trained to be predicted as true by testing an in-bound index. After training, an out-of-bounds index is used to *speculatively* execute the fourth line causing the data pointed by the attacker-controlled index to be cached. After the speculative execution, the value of `array1[untrusted_index]` can be known by scanning the cache blocks in `array2` using either Flush+Reload or Flush+Flush.

[24]There are other reliable measurement mechanisms for covert-channels [54]. Moreover, caches are not necessarily the only option for the covert-channel: Port contention [4] was demonstrated to work as well [5].
[25]This variant was also exploited remotely [64].

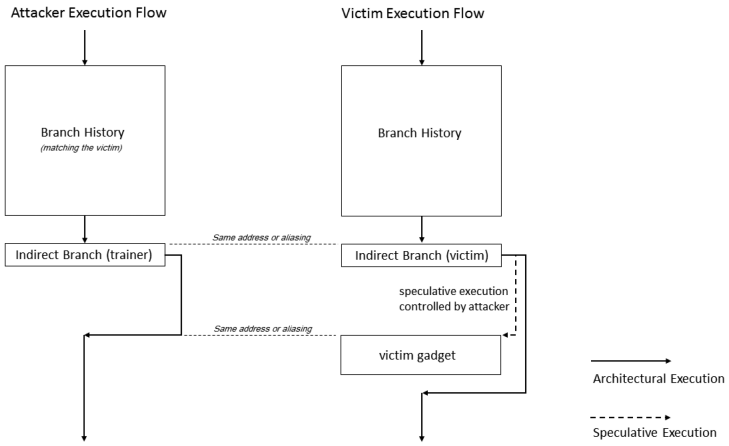


Fig. 12. Schematic of spectre variant II attack flow.

Note that in order to carry out the attack, the attacker needs to (1) induce latency in the conditional branch to create a window for speculative execution by evicting `array1_size` from the cache, (2) make sure the cache blocks for `array2` are evicted before the attack, and (3) perform direct or indirect access to `array2` to carry out the measurement.

Spectre Variant II targets the indirect branch predictor[26] and exploits the fact that it is shared across different security contexts and is not flushed upon a context switch. This allows an attacker to control the speculative execution of a victim from a different security domain by injecting target address entries into the indirect branch predictor. Figure 12 shows the attack flow of Spectre Variant II, where the attacker manipulates the speculative execution of an indirect branch in the victim by matching both its address and branch history. The desired target of the victim is a code gadget that performs data-dependent caching, which can be measured by the attacker.

Spectre has other attack scenarios, such as Application-to-Application, Application-to-OS, GuestVM-to-Hypervisor, OS-to-SMM, and so on. Given that different predictors are used for the different types of indirect branches, it was demonstrated that returns can also be used for attacks [53]. Intel proposed a new type of branch instruction to give control to software as a mitigation option [42]. The organization of the **Return Stack Buffer** (**RSB**) also changed based on the microarchitecture [43].

Although Spectre relies on speculative reads, speculative writes might also be used to control the speculation and are harder to detect and protect against [23]. Mechanisms such as hardware memory disambiguation and store-to-load forwarding also provide speculative writes. This essentially means that an instruction executed speculatively might "write" to a location and speculative reads during the speculative window will see the overwritten value instead of the committed value [40, 62, 71].[27] Intel also proposed a new memory type to make it easier to mitigate such issues [41].

Meltdown

Compared to the two Spectre variants, Meltdown exploits the behavior that speculative execution continues microarchitectural even when an architectural fault occurs. Therefore, a user mode code

[26]The indirect branch predictor stores the absolute or relative addresses of branch targets in an array, where each array entry is tagged by both the address of the indirect branch and the hash of the recent branch history.
[27]Memory disambiguation, also known as memory fusion, has been shown to have side-channels, as demonstrated in [57].

can try to directly load from a kernel address and make the subsequent cache load depend on the value read from the kernel space. This is shown in the code example below:

```
; rcx = kernel address
; rdx = measurement array
mov al, byte [rcx]
shl rax, 0xc
mov rdx, qword [rdx + rax]
```

Although the load in the third line will trigger an architectural exception due to the mismatched privilege level, with a proper setup such as using a branch that takes a long time to resolve, there is a window in which the execution continues speculatively before the exception is raised and the pipeline is flushed. In such a case, cache block loading that depends on the kernel space data (i.e., the fifth line) could already be carried out and not be reverted despite the exception. Afterward, using a registered user mode fault handler or a technology such as Intel TSX (see Section 3.5), the attacker can resume execution after the exception and carry out measurements in its own memory range to obtain the data from the kernel memory. The entire fault handling can be avoided if the load already occurs in a speculative channel, which can be created by the attacker using either variant of Spectre.

The L1TF attack has similar characteristics as Meltdown, but instead of a privilege violation fault, its vulnerability is triggered by a terminal page fault, which occurs when the page table entry for a virtual address is not present. An example L1TF scenario is a guest VM attacking the hypervisor by controlling the mapping between linear addresses and the guest's physical addresses, which is sufficient to gain control over the target of the leak. The leak occurs because the system will end-up using the guest physical address as if it is the host physical address during the speculative access. Therefore, if the contents of such host physical addresses are in L1 D-cache, the guest would be able to read them.

The mitigation proposed by Intel against Meltdown was a software change to split the address space between user mode and kernel mode [16], which is based on the proposed approach to protect against using prefetching to bypass KASLR (discussed in Section 4.7) [36]. The recommended fix for Spectre and L1TF involved a hybrid between software and microcode (i.e., new hardware capabilities implemented through new MSRs) [18]. Another option to protect against Spectre was proposed by Google that replaces all indirect branches for call/ret pairs and is named *retpoline* [15].

Note that the BTB side-channel attack discussed in Section 4.8 can be viewed as one of the earlier works that laid the foundation for the discovery of the speculative covert-channel attack. However, there are two major differences between these two types of attacks:

— A speculative covert-channel attack actively influences the speculative execution of the victim, while a BTB side-channel attack is passive where the attacker only observes the side-channel behavior of the victim's execution; and
— A speculative covert-channel attack causes the microarchitectural state to be leaked by forcing cache block loading. In contrast, a BTB side-channel attack uses measurement differences in the attacker to learn about the victim branch decisions.

4.9.1 The Special Case of Intel SGX. Intel **Software Guard eXtensions** (**SGX**) provides a secure (i.e., isolated) computing environment for user mode software called *enclave* that protects against attacks from the OS or other highly privileged software [14]. Nevertheless, side-channel attacks were suggested against it as soon as the technology was released [81]. All the different variants of the speculative covert-channels had a case/scenario to attack SGX, including [70, 71] and others.

The use of speculative execution was leveraged in [65] to quickly replay different possibilities and speed up brute-forcing against SGX. This attack does not require a restart of the target because speculative execution does not crash the target application and thus is much faster. This type of attack does not fit within the proposed taxonomy since the particularities are related to the speculative execution and not to the existence of a cache.

4.10 Side-channel on the Memory Management Unit

Page tables are essentially data stored in memory; therefore, accessing PTEs during a page walk will result in caching memory blocks containing the accessed PTEs. Since it is possible to probe the cache activity using side-channel attacks as mentioned in Section 4.4, there is an intrinsic cache-based side-channel that can reveal the actual virtual address being translated by the MMU, which is a Type 2 attack. The discovery of the virtual address being translated by the MMU breaks ASLR since the memory layout becomes known [34].[28]

There are two major factors that lead to the creation of this intrinsic side-channel:

— Page tables are page-aligned in memory and the page offset of a PTE directly correlates to certain bits of the virtual address being translated. For example, in current x86-64 platforms, the lower 48 bits of a 64-bit address are used for virtual memory addressing[29]: The lower 12 bits represent the page offset while the other 36 bits are the bits that are being translated. Assuming four levels of page tables are used as shown in Figure 2 and each PTE is 8 bytes in size, 9 of the 36 bits at each translation level determines the page offset of the corresponding PTE;

— In an N-way, set-associative cache with S sets with 64-byte block size, bits 6 to ($6+log_2S$-1) of the physical address determine which set the address maps to. Since physical and virtual addresses share the lower 12 bits as the page offset, bits 6–11 of the virtual address partially determines the address of the set. For example, suppose a cache has 8,192 sets, then the mapping is determined by bits 6–18 of the physical address. Bits 6–11, which are the same for both physical and virtual addresses, represent the page block index, and bits 12–18 of the physical address represent the *page color*. Therefore, there are 128 different page colors and two pages with the same color will have their cache blocks with the same page offset map to the same cache set.

Based on these two factors, the virtual address itself determines the page offset of the PTEs for different levels during the page walk and in turn, the cache sets loaded by the MMU. This could be measured by an attacker through a cache-based side-channel to recover the exact virtual address. This attack is also called *ASLR⊕Cache*, or simply *AnC* [34].

In the actual attack scenario published by Gras et al. [34], the attacker is inside the victim's browser sandbox with the aim of de-randomizing the memory layout to exploit memory corruption vulnerabilities. The attacker executes a JavaScript code to trigger a memory allocation of a predetermined size by, for example, creating ArrayBuffer (JavaScript) or spraying JITed code.[30] Since such a memory allocation is either page-aligned or starts from a known page offset, the attacker has control over the lower 12 bits of the accessed addresses. However, the attacker does

[28]This is different than the confused deputy [76] scenario proposed in [72], which breaks the isolation provided by page coloring.

[29]If the highest bits are not all the same, the access is named a "non-canonical" access. Non-canonical accesses should fault, but speculative non-canonical accesses were mis-handled by some systems, potentially leaking data [3] through a side-channel.

[30]JavaScript is JIT compiled to native code, and spraying means creating a large consecutive block.

not know the full virtual address because ASLR randomizes the bits 12 and above. This attack was demonstrated in a modern 64-bit Linux OS, which provides 28 bits of ASLR entropy.

As mentioned in Section 4.4, the Prime+Probe approach can be used to measure the cache set(s) accessed during the page walk of a specific translation. This requires the Prime stage to flush all TLB entries and page tables out of the cache by accessing a large eviction set. However, the eviction process itself may trigger a page walk and cache the accessed PTEs, which inevitably introduces noise to measurements.

To solve this problem, an alternative approach called *Evict+Time* is used. The Evict+Time approach is similar to Prime+Probe, but it only evicts one or one specific group of cache sets at a time and measures the execution time of the target routine. Evict+Time has less bandwidth, i.e., data are leaked slower than Prime+Probe, but it is more noise-resistant in this particular attack scenario.

In an AnC attack, an Evict+Time measurement involves the following three steps:

(1) Allocate a memory area as an eviction set that (a) contains pages with all the page colors and (b) has a number of pages larger than the number of entries in TLB;
(2) Evict a group of cache sets by accessing all the pages in the eviction set with page offset T. This will evict the targeted cache sets and also flush the D-TLB and the unified TLB (i.e., S-TLB); and
(3) Measure the time taken to access the target address V, which will trigger a page walk since Step 2 flushed the TLBs. The target itself is at a different page offset than T so that it does not get evicted. If any of the PTEs used in this page walk is hosted by the evicted cache sets, accessing the target address V will take a longer time, and vice versa.

The page offsets of the cache blocks hosting the PTEs of the target address V are known by repeating Steps 2 and 3 for all 64 possible T offsets, which would reveal the higher 6 bits of the 9 bits used in each level of translation. However, it is not clear at this point which offset corresponds to which translation level. To solve this, the attacker can slide the PTE at a specific level by moving the target address V with certain step sizes.

In a 64-bit 4-level translation (see Section 3.4), the PTL are referred to as PTL4 to PTL1 from high to low in the page table hierarchy. If the attacker shifts the target address V by an offset of $i \times 4$ KB with $i = 1, 2, \ldots, 8$, the PTE of PTL1 and only PTL1 will shift by a step of 8 bytes as i is increased. By observing when one of the cache block offsets becomes resolved from the changes in the original Evict+Time probing, the following two pieces of information can be obtained:

— The exact cache block offset corresponding to PTE of PTL1, which gives the higher 6 bits of the 9 bits translated at PTL1; and
— The smallest i that causes the shift in PTE cache block, which gives the lower 3 bits of the 9 bits translated at PTL1.

Therefore, all 9 bits of the target address V translated by PTL1 can be resolved. Similarly, the 9 bits for PTL2 can also be recovered using the same approach by changing the step size of V from 4 KB to 2 MB. For higher bits corresponding to PTL3 and PTL4, 8 GB and 4 TB of virtual memory need to be allocated, respectively, using the same sliding technique, which is not practical in this attack scenario. The authors resolved this challenge by allocating a 2 GB memory range and relying on the behavior of browser memory allocators to have the allocated region cross the boundary of PTL3 and PTL4.

Another level of complexity to the AnC attack is that the MMU may have a private page table cache that is separate from regular caches. The hardware implementation of the page table cache in MMU is architecture and platform specific and often undocumented. Since AnC attacks using Evict+Time rely on all levels of PTEs to remain only in regular caches, the page table cache of

MMU needs to be flushed (at least for a certain PTL) in order for this attack to work. This actually provides a good opportunity to reverse engineer the hardware design of MMU by developing an effective way of flushing page table cache with specially designed memory accesses [74].

4.11 Side-channel Attack empowered by the Integrated Graphics Processing Unit

Researchers are exploring creative ways to abuse cache issues. A recent example leverages the integrated GPU to not only accelerate microarchitectural side-channel attacks, but also to expose them to a remote attacker, e.g., from a browser by means of Java Script [31].

As with traditional cache-based side-channels, the reverse engineering efforts required due to the lack of documentation are still a barrier, but as more information is discovered, additional security issues will be found. For example, some new techniques to gather internal information on the GPUs, which have lead to the bypassing of existing CPU-based side-channel mitigations, have been discussed in [31].

4.12 Memory Performance Attacks

In 2007, Microsoft researchers unveiled a new class of attacks, dubbed *Memory Performance Attacks* [56]. Although this class of attacks is about memory access patterns, the root cause of the problem is the existence of a *row buffer* in the memory controller to cache the content of an entire memory row. The memory controller prioritizes memory requests to favor hits into the row buffer since it is faster. This possibility created a side-channel [60], but more importantly, the possibility of a denial of access to requests that do not hit in the row buffer. This is a Type 4 cache attack in which accesses to a cache are denied (or in this case, requests are slowed, since at some point it will be served).

They experimented with applications specifically causing access patterns that hit in the row buffer, and therefore their requests were prioritized causing a measurable slow down in the entire system execution.

4.13 Hardware Store Elimination

An interesting and subtle example of the influence of caches in security is the *Hardware Store Elimination* feature of the latest Intel client-based microarchitectures. This feature was undocumented until the release of a blog post showing performance improvements in the writing 0's versus 1's into memory [25]. While the blog in question discussed the performance results of writing of 0's versus 1's, security researchers immediately identified this differentiation as a side-channel.

The feature optimizes write operations for situations where the entire cache block is already filled with 0's (e.g., the blog demonstrates a *memcpy()* operation of a sequence of 0's in memory), which avoids complicated cache coherence states to track when it is evicted from one level of cache to another. But such optimization signifies that there is a clear and measurable difference between writing sequences of 0's and any other values. The consequence for security is not only on privacy where some content might be guessed by a different user on a system, but also on certain cryptographic implementations/requirements. For example, it is common to have operations with long sequences of zeros in cryptography, such as using transformation masks, having really long key sequences, or identifying trailing data in encrypted packets.

5 CONCLUSIONS

A modern computer architecture is complex and has many different components that influence the overall security. As a result of such complexities and ongoing microarchitectural enhancements, fully documenting all aspects of the architecture and components is not feasible. Thus, many of

the capabilities are not completely explained or there is not enough code to properly test all the inner workings.

This article discussed the general implications of cache behavior in modern platforms and provided a significant amount of technical depth on its impact on security. The challenges caused by having different configuration options and different implementation/design choices have historically created weaknesses and vulnerabilities, with the most relevant ones discussed in this survey. A lot of work still remains in this area and many other cache-related features were not discussed. However, the taxonomy together with case studies presented in this survey can be used as a guide to evaluating the impact of such features on the overall system security and over time can be expanded to cover new classes of issues.

ACKNOWLEDGMENTS

We would like to acknowledge the contribution from Gabriel Negreira Barbosa, who participated in the initial discussions and helped in the proof-of-concept code development, which was presented at Troopers Conference in Germany [6]. The code published was used to prove the behavior of different parts of caches in a given system, such as the cache disable bit and non-eviction mode. In addition, we would like to thank Kekai Hu who helped with the initial proofreading and many of the drawings. We would also like to thank Ke Sun for providing additional technical reviews and comments. We also would like to thank Joseph Nuzman for the deep technical review and feedback. Finally, we would like to thank Sergey Bratus, for the inspiring conversation that initiated one of the authors in the journey of hardware caches and their security implications, long before speculative side-channels were a thing.

REFERENCES

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. Association for Computing Machinery, New York, NY, 312–320. DOI: https://doi.org/10.1145/1229285.1266999

[2] Onur Acundefinediçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*. Springer-Verlag, Berlin, 225–242. DOI: https://doi.org/10.1007/11967668_15

[3] Inc. Advanced Micro Devices. 2021. Transient Execution of Non-canonical Accesses (CVE-2020-12965). Retrieved January 25, 2022 from https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1010.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. Cryptology ePrint Archive, Report 2018/1060. Retrieved from https://eprint.iacr.org/2018/1060.

[5] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. DOI: https://doi.org/10.1145/3319535.3363194

[6] Rodrigo Branco and Gabriel N. Barbosa. 2015. Modern Platform-Supported Rootkits. Retrieved February 05, 2017 from https://github.com/rrbranco/Troopers2015/blob/master/Troopers2015-Final-Presented-Public.pptx.

[7] BSDaemon, coideloko, and D0nAnd0n. 2008. System Management Mode Hack - Using SMM for Other Purposes. Retrieved February 22, 2017 from http://phrack.org/issues/65/7.html.

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, 249–266. Retrieved from https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking data on meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, 769–784. DOI: https://doi.org/10.1145/3319535.3363219

[10] Intel Corporation. [n.d.]. Microarchitectural Data Sampling Advisory. Retrieved December 01, 2021 from https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html.

[11] Intel Corporation. 2003. Intel Platform Innovation Framework for EFI. Retrieved May 05, 2021 from https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/efi-pei-cis-v09.pdf.
[12] Intel Corporation. 2009. 64-bit Intel Xeon Processor MP with 1 MB L2 Cache. Retrieved May 05, 2021 from https://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/30675212.pdf.
[13] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. Retrieved May 05, 2020 from https://software.intel.com/en-us/articles/intel-sdm.
[14] Intel Corporation. 2017. Intel Software Guard eXtensions. Retrieved April 30, 2020 from https://software.intel.com/en-us/sgx.
[15] Intel Corporation. 2018. Retrieved April 30, 2020 from https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation.
[16] Intel Corporation. 2018. Intel Analysis of Speculative Execution Side Channels. Retrieved July 10, 2018 from https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.
[17] Intel Corporation. 2018. Resources and Response to Side Channel L1 Terminal Fault. Retrieved August 18, 2018 from https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html.
[18] Intel Corporation. 2018. Speculative Execution Side Channel Mitigations. Retrieved July 10, 2018 from https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.
[19] Intel Corporation. 2019. Transactional Synchronization Extensions. Retrieved May 27, 2020 from https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-overview.html?language=en.
[20] Intel Corporation. 2020. 6th Generation Intel Processor Family Specification Update - Errata SKL046. Retrieved May 27, 2020 from https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/desktop-6th-gen-core-family-spec-update.pdf.
[21] Intel Corporation. 2020. Intel Data Direct I/O Technology Overview. Retrieved May 05, 2021 from https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf.
[22] Intel Corporation. 2020. Snoop Assisted L1D Sampling Advisory. Retrieved April 30, 2020 from https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00330.html.
[23] Microsoft Security Research & Defense. 2018. Analysis and mitigation of speculative store bypass (CVE-2018-3639). Retrieved April 30, 2020 from https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/.
[24] Mark Doran, Kevin D. Davis, and Mark Svancarek. 2009. UEFI Boot Time Optimization Under Microsoft Windows 7. Retrieved May 05, 2021 from https://www.intel.com/content/dam/doc/guide/uefi-boot-time-optimizaiton-windows7.pdf.
[25] Travis Downs. 2020. Hardware Store Elimination. Retrieved May 27, 2020 from https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html.
[26] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: A feasibility study. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, New York, NY, 8 pages. DOI: https://doi.org/10.1145/2768566.2768571
[27] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, Piscataway, NJ, 13 pages. Retrieved from http://dl.acm.org/citation.cfm?id=3195638.3195686.
[28] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016), 23 pages. DOI: https://doi.org/10.1145/2870636
[29] Anders Fogh. 2016. Cache side channel attacks: CPU Design as a security problem. Retrieved June 13, 2018 from https://conference.hitb.org/hitbsecconf2016ams/sessions/cache-side-channel-attacks-cpu-design-as-a-security-problem/.
[30] Anders Fogh. 2017. Negative Result: Reading Kernel Memory From User Mode. Retrieved July 05, 2018 from https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.
[31] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, et al. 2018. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. 195–210.
[32] Ge, Qian et al. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware.
[33] Ben Gras, Kaveh Razavi, Herbert Bos, et al. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 955–972.

[34] Ben Gras, Kaveh Razavi, Erik Bosman, et al. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the NDSS*. Retrieved from https://www.vusec.net/download/?t=papers/anc_ndss17.pdf.

[35] Daniel Gruss, Julian Lettner, Felix Schuster, et al. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, Vancouver, BC, 217–233. Retrieved from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss.

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, et al. 2017. KASLR is dead: Long live KASLR. In *Proceedings of the Engineering Secure Software and Systems*. Springer International Publishing, Cham, 161–176.

[37] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 368–379. DOI: https://doi.org/10.1145/2976749.2978356.

[38] Jann Horn. 2018. Exploiting Branch Target Injection. Infiltrate Security Conference, 2018.

[39] Jann Horn. 2018. Reading privileged memory with a side-channel. Retrieved January 03, 2018 from https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[40] Jann Horn. 2018. Speculative Execution, Variant 4. Retrieved April 30, 2020 from https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[41] Kekai Hu, Ke Sun, and Rodrigo Branco. 2019. A New Memory Type Against Speculative Side Channel Attacks. Retrieved April 30, 2020 from https://github.com/intelstormteam/Papers/blob/master/2019-A_New_Memory_Type_Against_Speculative_Side_Channel_Attacks.pdf.

[42] Kekai Hu, Ke Sun, and Rodrigo Branco. 2019. A New Type of Branch Instruction. Retrieved April 30, 2020 from https://github.com/intelstormteam/Papers/blob/master/2019-A_New_Type_of_Branch_Instruction.pdf.

[43] Kekai Hu, Ke Sun, and Rodrigo Branco. 2019. A Recursive Counter for Linked List RSB. Retrieved April 30, 2020 from https://github.com/intelstormteam/Papers/blob/master/2019-A_Recursive_Counter_for_Linked_List_RSB.pdf.

[44] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. 191–205.

[45] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 380–392. DOI: https://doi.org/10.1145/2976749.2978321

[46] David Kanter. 2010. Intel's Sandy Bridge Micro Architecture (Instruction Decode and uop Cache). Retrieved January 22, 2017 from http://www.realworldtech.com/sandy-bridge/4/.

[47] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, Bellevue, WA, 189–204. Retrieved from https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim.

[48] Paul Kocher, Jann Horn, Anders Fogh, et al. 2018. Spectre Attacks: Exploiting Speculative Execution. Retrieved July 05, 2018 from https://spectreattack.com/spectre.pdf.

[49] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCAT: Practical cache attacks from the network. In *Proceedings of the S&P*. Retrieved December 12, 2021 from https://download.vusec.net/papers/netcat_sp20.pdf.

[50] Nate Lawson. 2009. Side channel attacks on cryptographic software. In *Proceedings of the IEEE Security and Privacy*. IEEE. DOI: https://doi.org/10.1109/MSP.2009.165

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. 2018. Meltdown. Retrieved July 05, 2018 from https://meltdownattack.com/meltdown.pdf.

[52] Yangdi Lyu and Prabhat Mishra. 2018. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security* 2, 1 (2018), 33–50.

[53] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, 2109–2122. DOI: https://doi.org/10.1145/3243734.3243761

[54] Clementine Maurice, Manuel Weber, Michael Schwarz, et al. 2017. Hello from the other side: SSH over robust cache covert channels in the cloud. DOI: https://doi.org/10.14722/ndss.2017.23294

[55] Ronald Minnich. 2015. Coreboot Project. Retrieved April 30, 2020 from https://github.com/coreboot/coreboot/blob/4.1/src/cpu/intel/haswell/cache_as_ram.inc.

[56] Thomas Moscibroda and Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium on USENIX Security Symposium*. USENIX Association, Berkeley, CA, 18 pages. Retrieved from http://dl.acm.org/citation.cfm?id=1362903.1362921.

[57] Marco Oliverio, Kaveh Razavi, Herbert Bos, et al. 2017. Secure page fusion with VUsion: https://Www.Vusec.Net/Projects/VUsion. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 531–545. DOI: https://doi.org/10.1145/3132747.3132781

[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Proceedings of the CryptographersŠ Track at the RSA Conference*. David Pointcheval (Ed.). Springer, Berlin, 1–20.
[59] Jurgen Pabel. 2009. Frozen Cache. Retrieved February 05, 2017 from http://frozencache.blogspot.com.
[60] Peter Pessl, Daniel Gruss, Clémentine Maurice, et al. 2016. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, Austin, TX, 565–581. Retrieved from https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.
[61] Joanna Rutkowska. 2007. Beyond the CPU: Defeating Hardware Based RAM Acquisition. Retrieved February 05, 2017 from http://www.first.org/conference/2007/papers/rutkowska-joanna-slides.pdf.
[62] Michael Schwarz, Claudio Canella, Lukas Giner, et al. 2019. Store-to-leak forwarding: Leaking data on meltdown-resistant CPUs. arXiv:1905.05725. Retrieved from http://arxiv.org/abs/1905.05725.
[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the CCS*.
[64] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read arbitrary memory over network. arXiv:1807.10535. Retrieved from http://arxiv.org/abs/1807.10535.
[65] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, et al. 2019. MicroScope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, 318–331. DOI: https://doi.org/10.1145/3307650.3322228
[66] Sherri Sparks and Jamie Butler. 2005. Raising the bar for windows rootkit detection. *Phrack Magazine* (2005). Retrieved February 05, 2017 from http://phrack.org/issues/63/8.html.
[67] Frank Swiderski and Window Snyder. 2004. *Threat Modeling*. Microsoft Press.
[68] Jacob Torrey. 2014. MoRE shadow walker: TLB-splitting on modern x86. *Black Hat USA Conference Proceedings* (2014). Retrieved February 05, 2017 from https://www.blackhat.com/docs/us-14/materials/us-14-Torrey-MoRE-Shadow-Walker-The-Progression-Of-TLB-Splitting-On-x86-WP.pdf.
[69] Jim Turley. 2014. Introduction to Intel Architecture. Retrieved April 30, 2020 from https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf.
[70] Jo Van Bulck, Marina Minkin, Ofir Weisse, et al. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 991–1008.
[71] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proceedings of the 41th IEEE Symposium on Security and Privacy*.
[72] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, et al. 2018. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 937–954.
[73] Stephan Van Schaik, Alyssa Milburn, Sebastian Osterlund, et al. 2019. RIDL: Rogue in-flight data load. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers Inc., 88–105. DOI: https://doi.org/10.1109/SP.2019.00087
[74] Stephan van Schaik, Kaveh Razavi, Ben Gras, et al. 2017. *Reverse Engineering Hardware Page Table Caches Using Side-Channel Attacks on the MMU*. Technical Report IR-CS-51. Vrije Universiteit Amsterdam. Retrieved from https://www.vusec.net/download/?t=papers/revanc_ir-cs-77.pdf.
[75] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, 687–702. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan.
[76] Wikipedia. 2020. Confused Deputy Problem. Retrieved May 27, 2020 from https://en.wikipedia.org/wiki/Confused_deputy_problem.
[77] Wikipedia. 2020. IOMMU Hardware. Retrieved May 05, 2020 from http://en.wikipedia.org/wiki/IOMMU.
[78] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM Memory via Intel CPU Cache Poisoning. Retrieved February 22, 2017 from http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.
[79] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*. USENIX, Bellevue, WA, 159–173. Retrieved from https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu.
[80] Wenjie Xiong and Jakub Szefer. 2020. Survey of Transient Execution Attacks. arXiv:2005.13435. Retrieved from https://arxiv.org/abs/2005.13435.
[81] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 640–656. DOI: https://doi.org/10.1109/SP.2015.45

[82] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, 719–732. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[83] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, New York, NY, 305–316. DOI: https://doi.org/10.1145/2382196.2382230

Received 12 June 2020; revised 26 April 2022; accepted 2 May 2022