Security Concerns in Modern Micro-architectures

Nikhilesh Singh, Vinod Ganesan, and Chester Rebeiro
Indian Institute of Technology Madras
{*nik,vinodg,chester*}*@cse.iitm.ac.in*

Abstract

In the last two decades, the evolving cyber-threat landscape have brought to center stage the contentious tradeoffs between security and performance of modern microprocessors. The guarantees provided by the hardware to ensure no violation of process boundaries have been shown to be breached in several real-world scenarios. While modern CPU features such as superscalar, out-of-order, simultaneous multi-threading, and speculative execution play a critical role in boosting system performance, they are central for a potent class of security attacks termed transient micro-architectural attacks. These attacks leverage shared hardware resources in the CPU that are used during speculative and out-of-order execution to steal sensitive information. Researchers have used these attacks to read data from the Operating Systems (OS) and Trusted Execution Environments (TEE) and to even break hardware-enforced isolation.

Over the years, several variants of transient micro-architectural attacks have been developed. While each variant differs in the shared hardware resource used, the underlying attack follows a similar strategy. This chapter presents a panoramic view of security concerns in modern CPUs, focusing on the mechanisms of these attacks and providing a classification of the variants. Further, the authors discuss state-of-the-art defense mechanisms towards mitigating these attacks.

1 Introduction

For over half a century, microprocessor research has focused on improving performance. Various micro-architectural features such as cache memories, branch prediction, superscalar, speculative and out-of-order execution were developed to facilitate this. While some of these features, for example the cache memory, were introduced to hide the latency of slow components, others like branch predictors, helped hide overheads due to operations that slow down program execution. Features like out-of-order execution and speculative execution were introduced to better utilize available resources. Side-by-side, features were incorporated in processors to support better multi-programming. Features such as multi-core processors and hardware multi-threading were incorporated to allow multiple users to simultaneously share a processor. These features accelerated new computing paradigms, especially cloud computing, where multiple users simultaneously share common hardware, thereby drastically reducing computation costs.

A critical aspect of the cloud computing paradigm is the isolation between users. To isolate one user's program from another, security schemes such as protection rings, segmentation, page table access controls bits, virtualization support, hardware-based security, crypto-accelerators, and trusted execution environments were introduced. Very soon, it was realized that these security schemes were insufficient. The shared hardware became a source of information leaks that could undermine the isolation provided by the processor. These attacks, popularly known as *micro-architectural attacks*, made use of shared hardware resources to glean sensitive information such cryptographic keys, web pages visited, user passwords, and keystrokes. Different strategies such as time-driven attacks, Prime+Probe, Flush+Reload, and Evict+Time were proposed for this purpose. In a cloud computing environment, these attacks could leak information from one user to another, in spite of having all security features enabled.

In 2018, two potent micro-architectural attack variants were proposed, namely Meltdown [36] and Spectre [32], that exploited the speculative and out-of-order execution features present in microprocessors. These attacks leveraged the fact that a processor's speculation may not always be correct. When speculation goes wrong, the speculatively executed instructions, called transient instructions, needs to be discarded, and the CPU should be rolled back to a previous state. However, this rollback is not always perfect. The CPU would still have a reminisce of the transient instructions. Researchers showed how this reminisce can be used to leak secrets. These attacks, which came to be called *transient micro-architectural attacks*, could read the contents of any memory region, including the OS memory. It could also read memory from trusted enclaves, even though the enclaves used encrypted memory.

Since 2018, there been several variants of transient micro-architectural attacks including Zombieload [55], Foreshadow [12], Rogue In-Flight Data Load (RIDL) [58], Fallout [14], Load Value Injection (LVI) [13], and Crosstalk [49]. Each variant found a new vulnerability that could bypass isolation in the CPU. Many of these attacks are not easily prevented by software patches. For those that can, the patches have huge performance penalties. It would require fundamental changes in the CPU design to mitigate these attacks in hardware.

This chapter would provide an introduction to transient micro-architectural attacks. Starting from Meltdown and Spectre, the authors would dwell on the basic principle of the attacks. This would be useful in distinguishing between the various attack classes and discussing the available mitigation techniques. Section 2 provides a background of modern CPU micro-architecture and also gives an introduction to micro-architectural attacks. Section 3 discusses transient micro-architectural attacks and classifies them. Section 4 discusses the defenses for these attacks, while the final section has the concluding remarks.

2 Modern CPU Microarchitecture

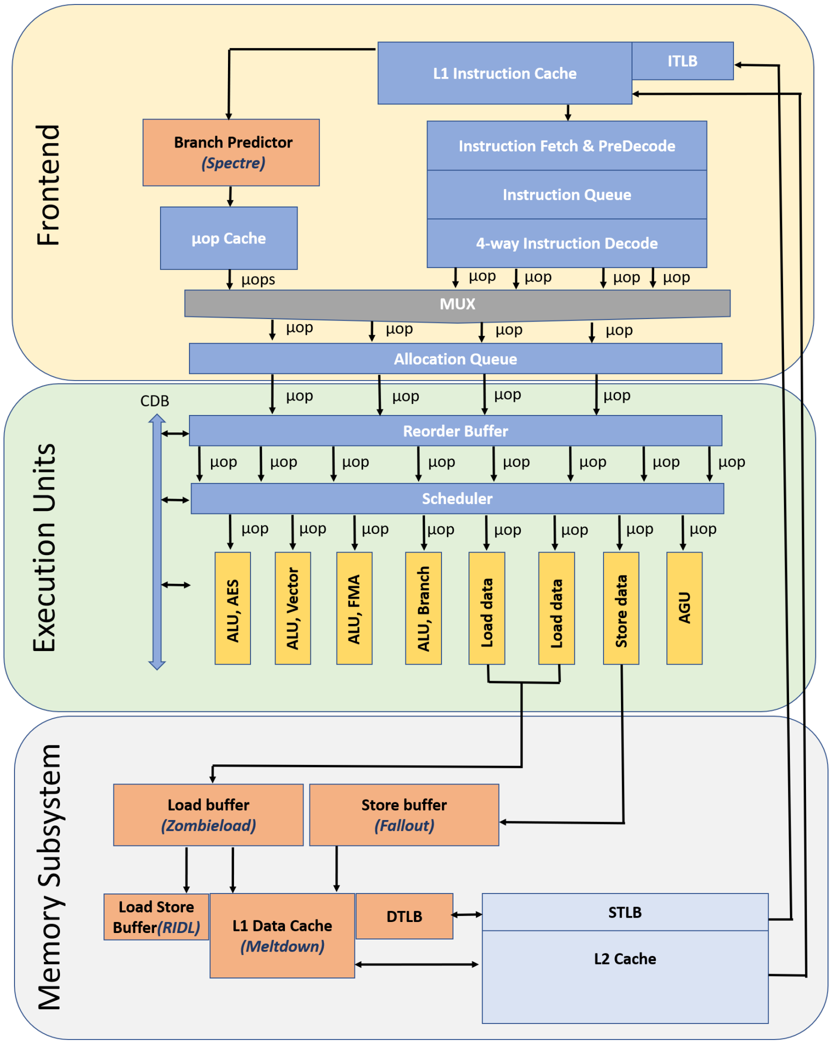


Figure 1: An Out-of-order Superscalar processor with vulnerable components shaded in Orange.

Notions of Security in Microprocesosrs. Beyond functional correctness, modern microprocessors attempt to enforce a root of trust to mitigate the ever-growing array of attacks. The goal of such approaches is to enable secure booting and provide platform to launch Trusted Execution Environments (TEE) post boot up. These TEEs, such as ARM Trustzone and Intel Software Guard Extensions (SGX) [53], ensure that the process boundaries guaranteed by the hardware are not violated by other processes. For example, the Intel SGX adopted in 2015, is a TEE feature supported by commercial processors that provide private regions of memory for programs. These regions are known as enclaves, cannot be accessed even from privileged software like the Operating System. This is achieved by encrypting enclave code and data present in the DRAM. Decryption is done when the code or data is fetched into the processor. Thus, the contents of an enclave, when in RAM, are always in an encrypted form and not accessible to any code outside the enclave regardless of the privilege levels. In recent years, however, researchers have shown that such trusted execution environments are not a panacea against the threat of transient micro-architectural attacks [12, 63]. The potency of these attacks is one of the reasons that led to the deprecation of Intel SGX from upcoming desktop processors [17, 18], posing further open questions regarding the security of hardware designs. In this section, we explore the premise of such attacks on the micro-architecture from first principles, starting with a background on the working of transient instruction in superscalar CPUs.

Transient Instructions in Superscalar CPUs. Figure 1 shows a block diagram of a superscalar CPU. In every clock cycle multiple instructions are fetched from the instruction cache into an instruction/decode buffer which forms the frontend. The instructions are decoded into a set of micro-ops and are continuously fed to the exeuction engines, such as Arithmetic and Logic Units (ALU) and Floating Point Units (FPU), through a dispatcher of allocation queues. The scheduler ensures that the issue is possible only if the functional unit is available and the operands used by the instruction are up-to-date. The instructions to the functional units can be issued out-of-order and based on a speculation, for example, the CPU can predict the outcome of a branch and speculatively execute instructions at the predicted branch target. The results from these speculatively executed instructions are stored in a temporary buffer and committed to registers and memory only when the speculation turns out correct. On the other hand, if the speculation is wrong, for example a branch is mis-predicted, the results from the speculatively executed instructions are dropped and not committed. These instructions are known as *transient instructions.* Besides branch mis-predictions there are several reasons that can cause transient instructions. For instance, a user-space program executing a load or store instruction from an illegal memory, for example from the kernel space, can result in a memory exception and also transient instructions. Another instance is of bounds check instructions that identifies if an index is within an array bounds. Memory operations following the bounds can be speculatively executed with any arbitrary out-of-bound index.

In addition to the out-of-order and speculative execution of processes, many modern CPUs support the execution of multiple programs simultaneously. This feature is known as Symmetrical Multi-threading (SMT). Instructions from two or more programs simultaneously execute in a single pipeline sharing hardware resources such as cache memories, branch prediction units, and various other on-chip resources.

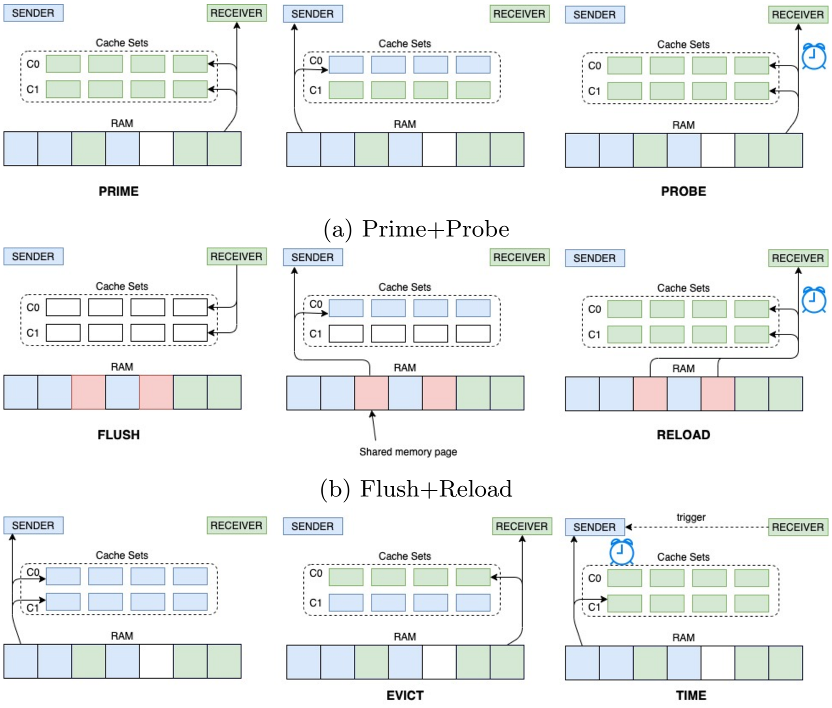
Micro-architectural State. As instructions flow through the CPU, various registers, buffers, caches and other memory structures in the CPU core store temporary data and results from the execution. While a few of these memory structures, for instance the general purpose registers, can be read or modified using instructions by instructions in the ISA, a significant portion of the structures are hidden and inaccessible from software. To enforce separation between applications, system software ensures that the data present in the ISA visible shared memory structures of one application cannot be read or modified by another application. For example, during a context switch, general purpose registers are either invalidated or loaded with the context of the next process that executes, thus achieving a temporal separation between the two processes. In multi-core or multi-threaded CPUs on the other hand, the ISA visible memory structures are duplicated enforcing spatial separation.

Unlike the visible structures, the hidden memory structures in the CPU, such as cache memories and branch prediction units are not always spatially and temporally separated between applications. They retain their values across context switches and are possibly shared in multi-core and multi-threaded CPUs. For example, a cache line that holds data from one application, can be evicted by another application. Similarly, a branch predictor trained on branches in one application, can influence the outcome of a prediction in another application. At first glance, this may seem innocuous as the structures are hidden from software. However, researchers have found that one application can indirectly affect another by these shared hidden memory structures. This has lead to a series of security vulnerabilities, commonly grouped in a category called micro-architectural attacks. The red regions in Figure 1 are modules in the processor with demonstrated security vulnerabilities. Researchers have used these vulnerabilities to break cryptographic algorithms, read Operating System data and break trusted execution environments.

Researchers have used these vulnerabilities to break cryptographic algorithms [6, 46], design keyloggers [50], fingerprint websites [57], break security features like Address Space Layout Randomization [5, 25, 28], leak sensitive information from the operating system [32, 36] and trusted enclaves [12, 63]. They have been applied on a variety of devices ranging from mobile phones to cloud computing servers. The next section provides a brief introduction to micro-architectural attacks.

2.1 Micro-architectural Attacks

This section introduces micro-architectural attacks using the example of cache memories. The cache memory is a high-speed memory placed between the CPU and RAM to cache recently used instructions and data. It can be simultaneously shared by multiple applications in a CPU core. Due to its small size, it can be the cause of contention when applications compete for the same cache line. The authors explain the fundamental working of micro-architectural attacks by using three examples. The first uses a prime and probe algorithm on a shared cache memory, while the second is an algorithm called flush and reload, that



(a) Prime+Probe

(b) Flush+Reload

(c) Evict+Time

Figure 2: Prime+Probe, Flush+Reload, and Evict+Time are the most common algorithms used to exfiltrate data in a micro-architectural attack. This figure demonstrates these algorithms in a covert channel that uses cache memory to transmit one bit of information from a sender to a receiver.

uses shared library code. The third is an evict and time algorithm on the cache memory.

Prime+Probe Attacks. Prime and probe forms for the basis for several micro-architectural attacks. It exploits the variance in the execution time caused by two applications that contend for the same shared hardware resource. The attack is discussed by showing an example of how the cache memory can be used to create a covert communication channel between a high-privileged application and a low-privileged application. Similar channels have been used on other shared resources as well, such as, TLBs, branch prediction units, load and store buffers, and even DRAM.

Consider that the high-privileged application called the *sender* and the low-privileged application called the *receiver* share a common cache memory. For example, the sender and receiver execute simultaneously on a CPU with a shared L1 data cache memory. The objective of the covert channel is to use the increase in execution time due to contention in the cache memory to transmit a message from the sender to the receiver. Apriori, the sender and receiver agree upon two cache sets $C0$ and $C1$ for communicating bit 0 and 1, respectively. The communication works as follows. **(1)** The receiver first performs memory load operations that fill both cache sets. This is called the *prime* phase and is done by loading data from addresses that map to sets $C0$ and $C1$ as shown in Figure 2a. **(2)** Depending on the message bit, the sender performs a memory operation to evict the receiver's data from the corresponding cache set. For example, to transmit a 0, the sender would evict the receiver's data from the cache set $C0$. **(3)** In the *probe* phase, the receiver repeats the memory operations in step (1), but this time also measures the execution time. Based on the execution time, the receiver can infer the transmitted bit since the memory access to the evicted cache set would take longer owing to the cache miss.

Prime+probe in micro-architectural attacks work similarly, except for the fact that the sender and receiver do not collude. Instead, the receiver primes sufficient number of sets in the cache (step (1)), waits for the sender to execute and evict one or more cache lines in these sets, and then performs a probe similar to step (3) to identify patterns in the sender's execution.

Flush+Reload Attacks. Unlike Prime+Probe attacks, where the information leakage is due to conflicts in the cache memory, in the Flush+Reload attacks, information leakage is caused without forcing cache conflicts. Consider, for instance the high and low-privileged applications sharing memory pages. Such sharing is common in systems that use shared libraries. A single copy of the shared library present in RAM is used by multiple applications. The time required to load data in a shared memory page depends on whether the data is in the cache or not. If present in the cache, the load will be considerably faster than if it is not present in the cache.

Consider that the sender and receiver of a covert channel decide on two shared regions of code or data, for example in a shared library. These regions are chosen so that they map to distinct cache sets: $C0$ and $C1$. In step **(1)**, the receiver ensures that the data in these two regions are not in the cache shown in Figure 2b. This is performed by a flush operation that evicts the addresses from the cache and is called the *flush* phase. On Intel x86 platforms, an instruction called `clflush` is used to perform this. The `clflush` takes an address as argument and flushes the addresses from all caches in the CPU. **(2)** In the second step, the sender, performs a load operations to either $C0$ or $S1$ depending on whether it wants to transmit a 0 or 1 respectively. It would cause the data from one of the two shared regions to be fetched into the cache. **(3)** The receiver then performs loads on both addresses and measures the time taken. This is called the *reload* phase. Only one of these two loads would result in a cache hit. The time when there is a cache hit would be much shorter than the time when there is a cache miss. This difference in time can be used to infer the bit transmitted. Unlike the Prime+Probe attacks techniques, Flush+Reload is independent of the cache attributes, like its associativity. It thus results in more portable attacks.

In transient micro-architectural attacks, the attacker defines an array. Similar to the covert channel, in step (1) the attacker ensures that no elements of the array is present in the cache memory. In step (2), the attacker triggers a transient load operation that forces exactly one element from the array to be loaded into cache. Similar to the step (3) in the covert channel, the attacker would do a reload to identify which element was loaded. In element of the array that is loaded transiently often reveal secret information, the Operating System data.

Evict+Time Attacks. Evict+Time attacks closely resemble the Prime+Probe attacks. The difference is that the adversary is able to accurately measure the execution time of the sender application. While this is a strong assumption, there are certain scenarios where such measurements are possible. For example, when the adversary can trigger the execution of the sender and an observable event marks the end of its execution. In such cases, the duration between the trigger and the event, serves as a measure of the execution time of the sender.

Consider again the covert channel between the high-privileged sender and low-privileged receiver application. The assumption at the start is that both cache sets, $C0$ and $C1$, have the sender's data. In the second step, the receiver evicts one of the cache sets, say $C0$ as shown in Figure 2c. In the third step, it triggers the sender to execute and monitors the execution time of the sender. The sender would transmit a 0 or 1 by loading data from memory that maps to the $C0$ and $C1$ cache set respectively. The time taken to perform this load differs for the 0 and 1 bit transmissions. transmitting 0 will result in a cache miss, thus experiencing a longer execution time compared to transmitting 1. This difference in time is observed by the receiver to infer the transmitted bit.

3 Transient Micro-architectural Attacks

When transient instructions execute, the hidden states of the CPU is modified. While the results of a transient operation is discarded after the speculation is proved wrong, the hidden state of the CPU is not rolled back. Thus, transient instructions have a permanent impact on the CPU state. Consider for example, the following code snippet.

```
I1. cmp r0, r1
I2. jne <dest_addr> /* branch to dest_addr, if r0 != r1 */
I3. mov r2, Addr1
I4. add r2, r1      /* r2  = r2 + r1 */
I5. load r3, r2     /* r3  = memory corresponding to (r2) */
```

In an out-of-order processor, instructions `I3`, `I4`, and `I5` can be transiently executed if the CPU mispredicts the branch outcome at `I2`. If the memory load in `I5` results in a cache miss, it causes data at the address present in `r2` to be loaded into cache. Due to the misprediction, the CPU would discard the results of instruction `I3`, `I4`, and `I5`; however, it would not roll back the state of the cache memory. Thus, data corresponding to the memory load in `I5` would

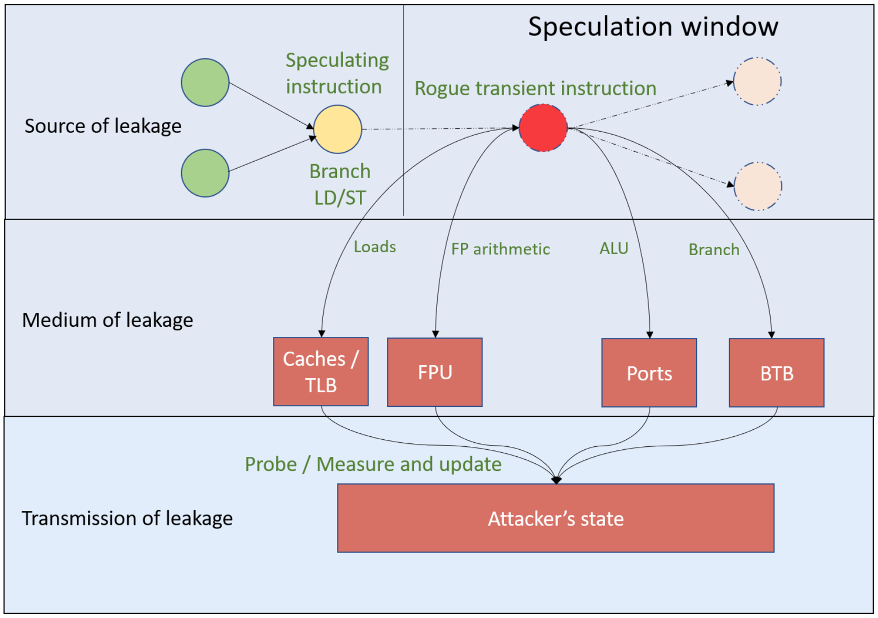


Figure 3: In a transient attacks, the transient instruction modifies the hidden states of CPU like cache memories, FPU, and ports, in a manner that depends on secret information. In the next stage, the attacker exfiltrates these secrets from the hidden states.

persist in the cache even after the transient executions are dropped. In 2018, researchers showed that this reminiscence of a transient execution could lead to serious security vulnerabilities that could potentially compromise every application executing on the CPU. The two attacks, namely Meltdown and Spectre, that were proposed in 2018 showed how this reminiscence could undermine the security of application software on a variety of commercial microprocessors. Since then, several variants of such transient attacks have been proposed. They form a new class of extremely powerful micro-architectural attacks and have come to be known as transient micro-architectural attacks or simply *transient attacks*.

A typical transient attack has three stages, as shown in Figure 3. The first stage disrupts the flow of program execution by forcing an exception or by inducing a misprediction that could trigger transient execution. In the next stage, the attacker relies on one or more of the transiently executed instructions to modify a hidden CPU state, such as the cache memory, branch predictor, or an internal buffer. The transient instruction is designed in a way so that the modification in the hidden state is correlated with a secret. The secret, for instance, can be keys of cryptographic ciphers, kernel code or data regions, or any other sensitive information. Due to the exception or the misprediction that occurred in the second phase, the transiently executed instructions are discarded,

Table 1: Transitive Micro-architectural Attacks

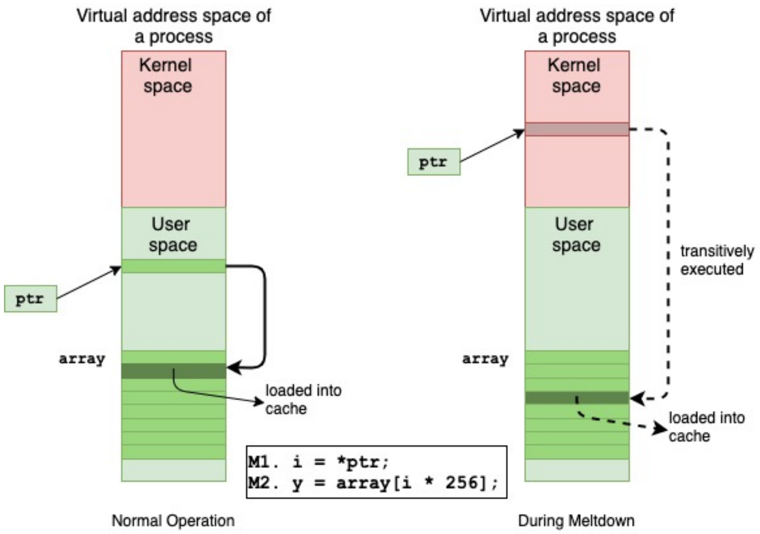
Attack	**Requirement**	**Source of Leakage**
—Meltdown and Spectre Like Attacks—		
Meltdown [36]	SMT	Transitive load
Spectre [32]		BPU
Foreshadow [12]		Transitive load
—Micro-architectural Data Sampling—		
RIDL [58]	SMT	Line feed buffer
Fallout [14]		Store buffer
Zombieload [55]		Line feed buffer
LVI [13]		Store buffer
Crosstalk [49]		Staging buffer

while the hidden micro-architectural states remain unaltered. In the final stage, the attacker exfiltrates information from the hidden micro-architectural state using an algorithm like Prime+Probe, Evict+Time, or Flush+Reload, to glean information about the secret.

After the initial attacks, vis-à-vis Meltdown and Spectre, several variants of transient attacks have appeared in the literature [8, 12, 13, 14, 15, 49, 55, 56, 58, 63]. Each new attack identified a new medium of leakage. Broadly, these attacks can be categorized into two classes based on the micro-architectural medium used for the leakage. The first is address-controllable transient attacks like Meltdown and Spectre, while the others are based on micro-architectural data sampling from internal buffers. While at a high level, the stages in both categories are the same and follow Figure 3, there are subtle differences between the two classes. Address-dependent attacks like Meltdown and Spectre use micro-architectural components like cache memories or branch prediction units as a medium for leakage. In these attacks, data (or instructions) placed in strategic memory addresses are transiently loaded (or executed). For example, in the covert channels described in Section 2.1, an address is used to select a cache set. The choice of the cache set is used as a medium for information leakage. In micro-architectural data sampling attacks like Zombieload and Crosstalk, on the other hand, it is not the address that is critical. Instructions are crafted so as to snoop into internal buffers like Re-order buffers, Line-Fill buffers, and load and store buffers. Table 1 classify the known attacks into these two categories.

3.1 Meltdown and Spectre like Attacks

These attacks require the knowledge memory regions of interest, and the attacker can target them specifically. Attacks like Meltdown [36], Spectre [32] and Foreshadow [12] fall in to this category. The upcoming sections look into each of these attacks to elaborate on their design and mechanisms.



```
M1. i = *ptr;
M2. y = array[i * 256];
```

Figure 4: Transitive execution of a memory load instruction causes data from `array` to be loaded into cache memory. Unlike the visible micro-architectural state, the cache contents are not rolled back when transient instructions are discarded. The contents of the cache can be gleaned using techniques such as Prime+Probe or Flush+Reload.

Meltdown. CPUs use protection rings to isolate privileged code. For example, Intel CPUs have four rings: Ring 0 to Ring 3. Privileged code, such as the Operating System's kernel, is assigned to Ring 0, while user processes are assigned to Ring 3. The hardware ensures that during regular operations, code executing in Ring 3 cannot read or write to Ring 0, thus isolating the kernel's code and data from userspace programs. The Meltdown attack exploits transient execution to read kernel data from a user program, thus breaching the isolation provided by the protection rings.

Prior to 2018, the kernel was mapped into the virtual address space of every process, as shown in Figure 4. This simplifies system calls and interrupt handling. Since the kernel was in Ring 0, a user function would not be able to directly access the kernel. The Meltdown attack showed how a userspace transient memory load or store operation to a kernel address caused the data to be loaded into the cache memory. This data could then be gleaned using one of the micro-architectural algorithms like Prime+Probe or Flush+Reload (Section 2.1).

In the first stage of Meltdown, the attacker writes code [43] as shown in Figure 4 that would perform a load from a kernel address. Specifically `ptr` is made to hold a kernel address. In the ideal case, this should have immediately created an exception because a user instruction is trying to read kernel data. However, modern CPUs are designed in a way that delays the exception, allowing subsequent instructions to be transiently executed. The contents of the kernel space data would thus be loaded into the register `i`, which is then used to load an element from the array into `y`. During this process, `y` is also stored in the cache memory. Notice that the array is indexed based on the kernel data. All of these instructions are transiently executed. At the time of throwing the exception, the CPU would discard the new values of `i` and `y`, but will not roll back the cache memory.

In the final stage of Meltdown, either the Flush+Reload or the Prime+Probe can be used to identify the cache set that holds the loaded `array` data, thus revealing information about the kernel data. With the Flush+Reload, for instance, the attacker would first ensure that all `array` elements are flushed from the cache before the transient instructions `M1` and `M2` execute. Post their execution, exactly one element corresponding to `y` would be present in the cache. The cache set that holds `y` can be inferred by measuring execution time to load each `array` element. The cache set containing `y` would have the shortest load time due to a cache hit. All other elements, by virtue of the initial flush, would result in cache misses.

Spectre. While the Meltdown attack makes use of an illegal load or store memory operation to induce a transient execution, Spectre makes use of mis-predicted branches. Modern microprocessors have a Branch Prediction Unit (BPU) that speculates the direction and the target address of a branch during program execution. The prediction is done by learning patterns in taken and not-taken branches from the branch history. For example, consider the following code snippet, where `array1_size` is the size of `array1` and is used to check the bounds of the index `x`. Statements `S2` and `S3` are executed only if `x` is within bounds.

```
S1. if (x < array1_size){
S2.    i = array1[x];
S3.    y = array2[i * 256];
S4. }
```

If the snippet is executed repeatedly with legal values of `x`, the BPU would learn the execution pattern and speculatively execute statements `S2` and `S3`. The results in `i` and `y`, however, would be committed only after the check `x < array1_size` is completed. After a while of such repeated executions, if `x` is made illegal (*i.e.* `x` $\geq$ `array1_size`), the BPU would predict incorrectly leading to transiently executed `S1` and `S2`. The two transient memory operations would load data into cache. The mis-prediction would ignore the new values computed for `i` and `y` but not rollback the cache memory. The final stage of Spectre is similar to Meltdown and uses micro-architectural attack techniques like Evict+Time and Flush+Reload to glean information about `array1[x]` from the cache memory. For example, if `array1[x]` corresponds to a kernel region, the attack would reveal the contents of the kernel location.

Spectre is one of the most powerful of all transient attacks because it is very difficult to mitigate. Over the years, multiple variants of Spectre have been proposed that exploit the different components of branch speculation in the processor. The different variants of Spectre attempt to tune different tables in the BPU. For example, [32, 56] exploits the Path History Table (PHT), while [8, 15, 32] exploits the Branch Target Buffer (BTB), and [33, 39] use the Return Stack Buffers (RSB).

Foreshadow. The Meltdown attack breaches the isolation provided by CPU's protection rings there by reading kernel data from a user program. Since 2015, Intel has added another level of protection in its processors. The Intel Security Guard Extension (SGX) is a feature supported by commercial processor variants (deprecated, 11th generation Intel Core onwards [17, 18]) that provide private regions of memory, called enclaves, for programs. It is ensured that the contents of an enclave, when in RAM, are always in an encrypted form, barring any access to a piece of code outside the enclave, regardless of the privilege levels.

The Foreshadow attack makes use of the fact that data in the SGX enclaves are stored in the plain form in the L1 cache. This allows transient instructions to compute on the cached secrets. The challenge is to cache secret data in the enclave and use them in transient operations. Given this, the Foreshadow works very similar to Meltdown [36]. It uses a local buffer that is transiently accessed at indices that depend on secret data stored in the enclave. Now that the entries from the buffer are in the cache, the attacker simply deploys the Flush+Reload technique to establish the secret from the enclave.

In principle, Foreshadow attacks are a variant of the Meltdown attack that use the same vulnerability, not just to read kernel memory from user space, but to rupture security mechanisms Intel SGX [53] that attempt to provide secure enclave protection domains. An improvement on the basic attack is Foreshadow-NG (Next Generation) [63] which has the potential to read any information that comes to the L1 cache affecting Virtual Machines (VMs), hypervisors (VMM), operating system (OS) kernel memory, and System Management Mode(SMM) memory.

3.2 Micro-architectural Data Sampling Attacks

Supporting speculative and out-of-order execution in a microprocessor often requires buffers at several locations in the CPU pipeline that temporary stores details about in-flight instructions. For example, Reorder Buffers (ROBs) are used to track instructions executed out-of-order and commit their results in the correct order. Other examples are the store buffers, used to track pending stores involved in optimizations. Micro-architectural Data Sampling (MDS) attacks are able to snoop into these temporary buffers to glean secret data from other applications. Unlike Meltdown and Spectre like attacks, MDS attacks are not tied to specific memory addresses, making it almost impossible to mitigate from software. This section summarizes the known MDS attacks.

Rogue In-Flight Data Load (RIDL). In traditional cache memories, a cache miss would block any further memory requests until the cache miss is serviced.

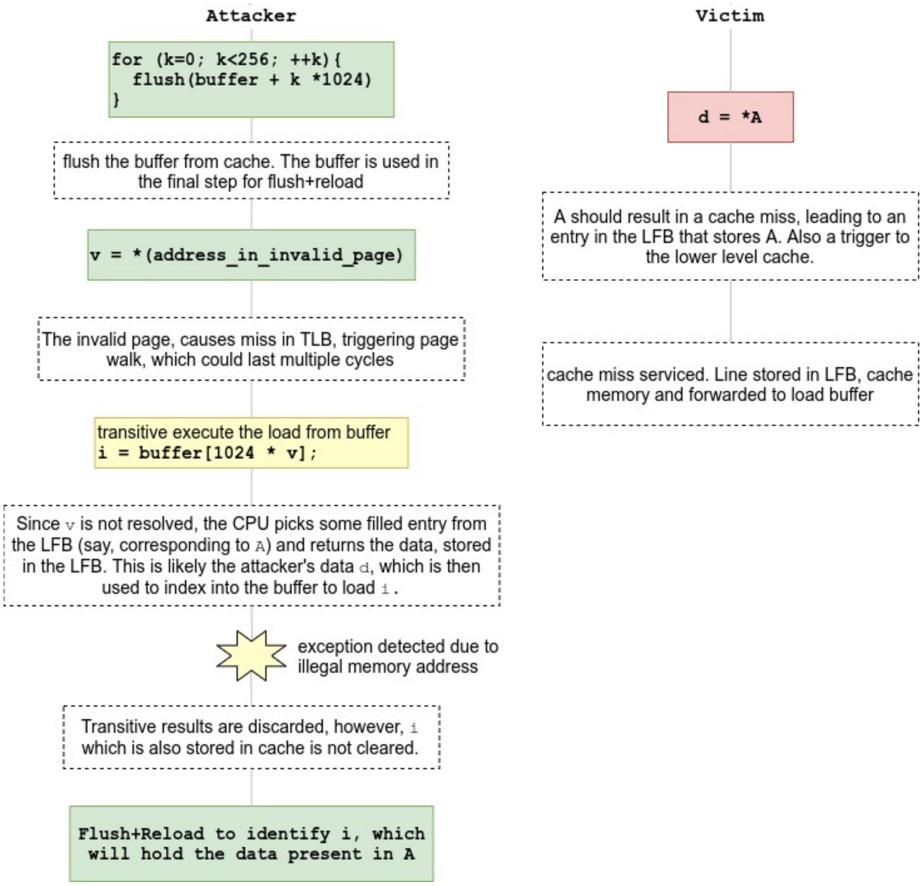

Attacker
Victim
for (k=0; k<256; ++k){
flush(buffer + k *1024)
}
flush the buffer from cache. The buffer is used in the final step for flush+reload
v = *(address_in_invalid_page)
The invalid page, causes miss in TLB, triggering page walk, which could last multiple cycles
transitive execute the load from buffer
i = buffer[1024 * v];
Since v is not resolved, the CPU picks some filled entry from the LFB (say, corresponding to A) and returns the data, stored in the LFB. This is likely the attacker's data d, which is then used to index into the buffer to load i .
exception detected due to illegal memory address
Transitive results are discarded, however, i which is also stored in cache is not cleared.
Flush+Reload to identify i, which will hold the data present in A
d = *A
A should result in a cache miss, leading to an entry in the LFB that stores A. Also a trigger to the lower level cache.
cache miss serviced. Line stored in LFB, cache memory and forwarded to load buffer

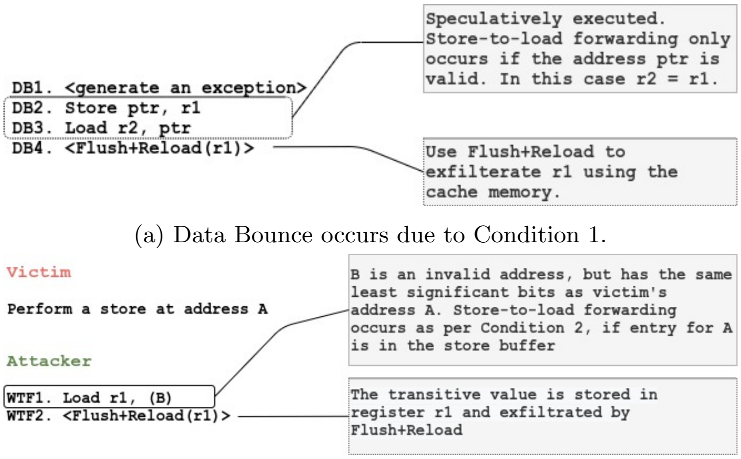
Figure 5: In the RIDL attack, the attacker (in green) snoops into the Line Fill Buffer (LFB) to read the victim's sensitve data present in the address (A).

In out-of-order CPUs, addresses corresponding to cache misses are stored in a Line Fill Buffer (LFB), so that subsequent memory requests can be serviced. This helps create a non-blocking cache. On receiving a memory request that results in a cache miss, an entry in the LFB is created to store the requested address. Subsequently, when the memory block is fetched, it is stored in the LFB entry corresponding to the memory address. The block is also stored in the cache memory and forwarded to the CPU core. The RIDL attack is able to snoop into the Line Fill Buffer (LFB) to retrieve the data from the stored block. Interestingly, the attack does not depend on the address of the memory request, but only requires a cache miss that makes an entry in the LFB.

RIDL assumes that the attacker and victim share a common L1 cache memory. The steps of the attack are shown in Figure 5. The attacker first ensures that `buffer` is flushed from cache and then triggers the victim to execute a load instruction, say at address `A`. If this victim's load results in a miss in the L1 cache, a new entry would be created in the LFB which would store the physical address of `A`. The attacker, running on a different thread in the same core, issues a load to an address present in a new invalid page. Since this page is new, it would result in a TLB miss and trigger a page table walk. The CPU would eventually detect that the load request is from an invalid page and mark it for exception. The exception is however thrown much later when the operation's results are committed in-order. During this time, the memory load operation from `buffer[1024 * v]` would continue transitively using an arbitrary value of `v` picked from an entry in the LFB. The address parts of the LFB entry is not matched, therefore, with significant probability, the entry would correspond to the victim's load request at `A`, resulting in `v` holding the value of the victim's data `d`. Thus `buffer[1024 * v]` is indexed at a location that is dependent on `d`. The result, is stored in `i`, as well as in a cache set. After the exception is thrown due to the illegal address, the transitive results in `v` and `i` are discarded, however, the cache is not rolled back. Flush+Reload is then used to identify `i`, thus revealing information about the attacker's data.

Zombieload. This attack [44], exploits LFB like RIDL, some unknown micro-architectural components and the concept of micro-code assists to mount the attack. Recall that an LFB tracks all load values that are not present in the L1 data cache and needs servicing from higher-level cache hierarchies. Whenever there are complex micro-architectural conditions, such as page-faults, it can be handled in one of two ways: (i) The fault can be delegated to a software service routine, or (ii) One can employ *microcode assists*, where the fault is handled through a set of microcode routines, which is faster than delegating to a software. A microcode assist always triggers a pipeline flush resetting the architectural state. However, in-flight instructions still finish execution only to be discarded later. Similarly, the outstanding LFB entries are not discarded. To not incur additional delays in completing the execution of in-flight instructions, the LFB is allowed to load *stale values* for previous load or store instructions, altering the micro-architectural state and potentially allowing the leakage of data. This data can be gleaned by the process of data-sampling explained above.

Though this attack looks similar to RIDL, the key contrast of this work is that the above leakage occurs even if the authors systematically ensure using Intel TSX [29] that there is no entry filled in Line Buffer during a cache miss. Intel TSX is a set of hardware extensions that enable one program or a program-thread to acquire a lock on certain memory locations in the memory which is prohibited from being updated or used by any other program until released. This enables concurrent programming as the updates in these locations are done atomically by one program or a thread at a time. Within a TSX window, and during certain situations, a miss in the L1-cache never creates a line-buffer entry. However, even without LFB, the leak happens, rather surprisingly at a much higher rate. This suggests that Zombieload is working not only because of LFB but also due to other unknown micro-architectural components, such as FPU register file and store buffer.



```
DB1. <generate an exception>
DB2. Store ptr, r1
DB3. Load r2, ptr
DB4. <Flush+Reload(r1)>
```

Speculatively executed. Store-to-load forwarding only occurs if the address ptr is valid. In this case r2 = r1.

Use Flush+Reload to exfilterate r1 using the cache memory.

(a) Data Bounce occurs due to Condition 1.

Victim

```
Perform a store at address A
```

Attacker

```
WTF1. Load r1, (B)
WTF2. <Flush+Reload(r1)>
```

B is an invalid address, but has the same least significant bits as victim's address A. Store-to-load forwarding occurs as per Condition 2, if entry for A is in the store buffer

The transitive value is stored in register r1 and exfiltrated by Flush+Reload

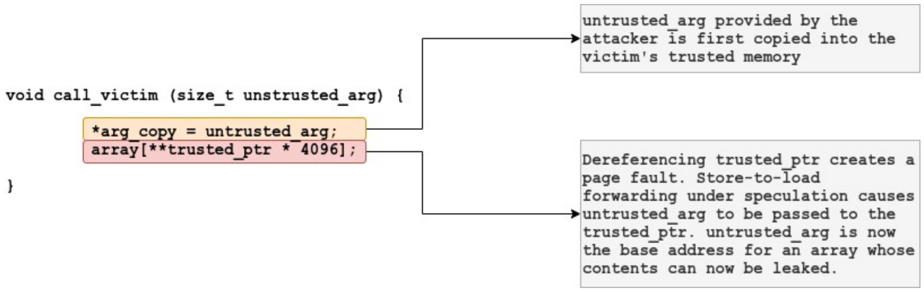
(b) Write Transitive Forwarding Vulnerability occurs due to Condition 2.

Figure 6: Fallout makes use of Store-to-Load forwarding of data in the store buffer to a speculatively executed load operation. The load operation can be from a different security domain, for example the kernel. The result of the load is stored in the `r1` register and exfilterated using Flush+Reload. Flush+Reload is similar way to previous attacks. The flush is done before the exception causing instruction, while the reload is done after the transitive execution is discarded.

Fallout. Out-of-order processors hide the latency associated with store operations by using a store buffer. On encountering a store operation, an entry is created in the buffer to hold the virtual address, physical address, and the value to be stored in memory. After the entry is created, subsequent operations in the program can speculatively execute permitting the stores to complete asynchronously. If one of the subsequent operations is a load, the data from the store buffer is forwarded. This is called *store-to-load* forwarding. Such *store-to-load* forwarding is possible in two conditions:

- **Condition 1.** If the complete address in the load matches the complete address of an entry in the store buffer, then the value in the entry can be directly used.
- **Condition 2.** If the virtual to physical address translation for the load fails, and a few least significant bits match with an entry in the store buffer, then the value in the entry can be speculatively used.

In their paper [14], the authors show how both these conditions can lead to transient attacks. The attacks arise from the fact that store-to-load forwarding can happen across security domains. It only requires either of the two conditions to be met. For example, the value in the store buffer entry will be forwarded just by matching address bits in the store buffer entry and the load operation. The store could be from the kernel, while the load from a user-space program.



```
void call_victim (size_t unstrusted_arg) {

        *arg_copy = untrusted_arg;
        array[**trusted_ptr * 4096];

}
```

untrusted_arg provided by the attacker is first copied into the victim's trusted memory

Dereferencing trusted_ptr creates a page fault. Store-to-load forwarding under speculation causes untrusted_arg to be passed to the trusted_ptr. untrusted_arg is now the base address for an array whose contents can now be leaked.

Figure 7: In LVI, the attacker injects a malicious value through load forwarding and uses that to leak sensitive data.

The second condition, leads to an attack called *Data Bounce*, that is used to identify if a virtual address is valid (*i.e.* mapped to a physical address). The pseudo-code is shown in Figure 6a. This attack can be used to break Address Space Layout Randomization (ASLR) [1, 7, 66]. The first condition, leads to a vulnerability called *Write Transient Forwarding (WTF)*. The vulnerability can be used to snoop into stores from another process. Figure 6 provides more details about these attacks.

Load Value Injection (LVI). In traditional Out-of-order processors, a store to a memory-location followed by a subsequent load instruction to the same memory-location can be slow as it comprises of two sequential instruction executions involving costly memory accesses. However, a widely used optimization to alleviate this, as explained above in fallout, is to perform *store-to-load* forwarding that will forward the contents of the producing store directly to the consuming load if both the entries are present in the load/store buffer. However, the effective addresses of the load / store instructions are not resolved until later and hence they are speculated instead. Therefore, during speculation there is a possibility that a wrong store forwards a value to the load. LVI uses this key principle to poison the data that the victim operates on to leak information. This is illustrated using Figure 7. In this example, the `untrusted_arg` is sent by the attacker which the victim stores within its buffer space (trusted memory), termed as the poisoning phase. Now, in case there is a page-fault caused when dereferencing trusted_ptr, the `trusted_ptr` erroneously receives value from the `untrusted_ptr` due to store-to-load forwarding within the load-store buffers under speculation. This poisoned data is now the index variable for an array whose values can now be leaked through standard cache-based attacks such as Flush + Reload. Generally, the attack contains three phases: (i) Micro-architectural poisoning where the attacker prepares the injection of a poison value by loading that in one of the micro-architectural buffers, (ii) The attacker then provokes the victim into executing instructions that cause a page fault or exception which triggers this store-to-load data poisoning. This can be done, for instance, by evicting a set of victim's virtual memory pages, and (iii) Gadget-based secret transmission, where the attacker finds exploitable code gadgets that can leak data under incorrect transient execution behavior and lead the victim to that code-gadget by carefully poisoning the data.

Crosstalk. Crosstalk demonstrates that MDS vulnerabilities exist beyond the CPU core through a shared memory buffer, called staging buffer, that is shared across multiple CPU cores. The authors identify several micro-instructions that touch the buffer. These instructions, if executed transiently, can potentially lead to leakage from one CPU core to another. One usecase of Crosstalk is to leak hardware generated random numbers that uses Intel's Secure Key Technology. The Secure Key technology makes use of an off-core hardware random number generator. The generator is initialized using the `RDSEED` instruction and the random numbers are read using the `RDREAD` instruction. These form the basis of several cryptographic primitives including, Intel's security enclaves. Executing either of these instructions touches the staging buffer. MDS attacks can be mounted on the buffer by transiently executing `RDRAND` and `RDSEED` thus leaking the seed or the random numbers generated by the hardware random number generator.

4 Countermeasures

Since their discovery, there have been extensive efforts to design develop countermeasures for transient micro-architectural attacks. The countermeasures can be broadly classified as prevention-based or detection-based. Prevention-based solutions attempt to stop the attack by thwarting the execution at one of the three phases (refer Figure 3). Naïve preventive solutions, for instance, disable speculative execution, thus preventing any transient execution, the first stage of the attack. Another naïve preventive solution disables all timers, thus preventing timing channels. This would disable stage 3, *i.e.* the transmission of leakage. In contrast, detection-based solutions do not disable any feature, rather, they aim to identify patterns in the program execution that can be attributed to an attack. While preventive-based solutions have high overheads, detection-based solutions suffer from false positives. Over the last few years, there have been multiple detection-based and preventive-based solutions proposed. Table 2 provides a list of these solutions. This section provides a description and analysis of some of these existing solutions.

4.1 Prevention-based Countermeasures

Figure 3 shows the stages of a transient attack. The attacker first identifies a source of leakage, as listed in Table 1. The next step involves the transient movement of data from the source to the medium of leakage. Finally, the attacker uses techniques established in Section 2.1 to transfer the secret information from the medium. Thwarting any of these sequential stages is sufficient to prevent the attack. Different preventive countermeasures target attacks at different stages of their execution, as described in Table 2.

Table 2: Countermeasures for Transient Micro-Architectural Attacks are classified as either prevention-based or detection-based. While prevention-based techniques aim to either modify or disable some functionality in the software or hardware, detection-based techniques rely on accurately identifying attacks from their run-time characteristics. [HW: Hardware implementation, SW: Software Implementation]

Stage of applicability	**Paper**	**HW or SW?**	**Threat Model**	**Reported Overheads**
–Prevention-based–				
Source of Leakage	NDA [62]	HW	Speculative execution attacks	4-32%
	Context [54]		Spectre-like	0-338%
	InvisiSpec [67]		Spectre-like	5-17%
	Safespec [30]		Meltdown, Spectre-like	3%
	SpectreGuard [23]		Spectre-like	8-20%
	Specshield [4]		Speculative execution attacks	21%
	Spectrum [24]		Spectre-like	2%
	MuonTrap [2]		Spectre-like	4%
	Invisible Speculation [51]		Cache and memory side-channels	11%
	Reversispec [65]		Speculative load attacks	8.3%
Medium of Leakage	Random-fill [37]	HW	Contention and reuse based attacks	Negligible
	Newcache [38]		Contention and reuse based attacks	Negligible
	CEASER [47]		Contention-based attacks	1%
	Encrypted-address cache [48]		Contention-bases attacks	1%
	Scattercache [64]		Cache leakage techniques (Section 2.1)	2-4%
	DAWG [31]		Cache timing attacks	4-7%
	SecDCP [60]		Timing side-channels	12.5%
	MI6 [10]		Spectre-like	16.4%
Transmission of Leakage	Timewarp [40]	HW	Timing side-channels	Negligible
	InvarSpec [69]	SW	Speculative execution attacks	[67]: 10.9%
	oo7 [59]	SW	Spectre-like	5.9%
	SPECCFI [34]	SW	Spectre-like	1.9%
–Detection-based–				
Transmission of Leakage	Cyclone [26]	SW	Cache leakage techniques	3.6%
	[16]		Cache leakage techniques	-
	NIGHTs-WATCH [41]		Cache leakage techniques	2%
	WHISPER [42]		Cache leakage techniques	-
	[3]		Cache leakage techniques	-
	CloudRadar [68]		Cross VM attacks	5%
	CacheShield [11]		Cross VM attacks	-

Prevention-based countermeasures provide a preemptive solution to these attacks. While the goal of all solutions is to disable potentially vulnerable behavior of programs, they differ in the attack phase they target. For example, a preventive solution, called TimeWarp [40] fuzzes the timers in order to prevent attackers from making fine-grained measurements. Such fine-grained measurements are needed to distinguish between micro-architectural events like cache hits and misses. Without precise time measurements, the third phase of the attack, namely the flush+reload, would fail. While most of these solutions are implemented in the hardware, there are also proposals that work from the software [34, 59, 67].

Prevention at the source of leakage. These countermeasures attempt to thwart attacks at the source of leakage. The most popular approach is to redesign speculative execution in processors to make it leakage-free. A typical solution in this direction divides load instructions into safe and unsafe categories based on the threat model. For example, a load instruction that has committed its results can be considered safe, while a speculative load that is yet to be completed is considered unsafe to prevent Meltdown and Spectre-like attacks. Countermeasures designed on this philosophy allow the safe loads to alter the global state of the caches. Unsafe loads, however, are not allowed to affect the state of the cache hierarchy [2, 4, 24, 65, 67]. To implement this, a buffer is inserted in the processor design that temporally holds the results from speculatively executed instructions until the instruction is completed.

Prevention at the medium of leakage. Cache memories store a subset of data in the memory based on temporal and spatial locality. As the cache is several times smaller than the main memory, multiple addresses map to the same location in the cache resulting in contention. The contention is possible within a process and also across processes. An attacker models this contention to glean information in the cache, using techniques seen in Section 2.1. Specialized cache memory designs have been proposed for thwarting the attacks by reducing cache contention.

In [46], Percival suggests eliminating cache contention by modifying the cache eviction algorithms. The modified eviction algorithms would minimize the extent to which one thread can evict data from another thread. In [45], Page proposed to partition a cache memory that was built of direct-mapped cache sets that could dynamically be partitioned into protected regions by the use of specialized cache management instructions. By tagging memory accesses with partition identifiers, each memory access is hashed to a dedicated partition. While this prevents cache contention from multiple processes, the cache memory is under-utilized due to rigid partitions. For example, a process may use very few cache lines of its partition. The unused cache lines are not available to another process.

In [61], Wang and Lee provide an improvement on the work by Page [45] using a construct called *partition-locked cache* (PLCache), where the cache lines of interest are locked in the cache, thereby creating a private partition. These locked cache lines cannot be evicted by other cache accesses not belonging to the private partition. In the hardware, each cache line requires additional tags comprising of a flag to indicate if the line is locked, and an identifier to indicate the owner of the cache line. The under-utilization of Page's partitioned cache still persists because the locked lines cannot be used by other processes, even after the owner no longer requires them.

In [22], Domnitser et al. provide a low-cost solution to prevent attacks based on the fact that the cipher evicts one or more lines of the spy data from the cache. The solution, which requires minor modifications of the replacement policies in cache memories, restricts an application from holding more than a pre-determined number of lines in each set of a set-associative cache. With such a cache memory, the spy can never hold all cache lines in the set, therefore the probability that the cipher evicts spy data is reduced. By controlling the number of lines that the spy can hold, a tradeoff between performance and security can be achieved. Over the years, several other cache partitioning techniques have been suggested [31, 52] which strengthens the defense while improving usability.

Another well-known modification defense for cache-based attacks makes use of randomization. Wang and Lee propose a *random-permutation cache* (RPCache) in [61], whereas the name suggests, randomizes the cache interference to make the attack more difficult. The design is based on the fact that information is leaked only when cache interference is present between two different processes. RPCache aims at randomizing such interferences so that no useful information is gleaned. The architecture requires an additional hardware called the *permutation table*, which maps the set bits in the effective address to obtain new set bits. These are then used to index the cache set array. Changing the contents of the permutation table will invalidate the respective lines in the cache. This causes additional cache misses and randomization in the cache interference.

An advancement of random cache architectures are designs that encrypt the mapping of addresses to cache sets. CEASER incorporates a block cipher [47, 48] for performing the encryption. The encryption key is periodically changed to obtain a different mapping for the cache sets. An important aspect of this design is the encryption algorithm, since it lies in the critical path and influences the time for load and store operations. While traditional ciphers have considerable latencies, ciphers designed specifically for this purpose may not provide sufficiently strong encryption [9].

Prevention at the transmission of leakage. While there are several techniques to thwart transient attacks by modification in the cache and the execution, existing literature also includes some preventive solution that aims to target the root cause of timing channels, such as fuzzing the timer [40] or increasing the entropy [21] in the timing information. There also solutions that use program analysis [59, 69] on the program code to identify vulnerable regions and forbid speculative execution in those code sections [23].

4.2 Detection-based Countermeasures

Unlike prevention-based countermeasures, detection-based solutions tend to be reactive in their approach. The detection of any micro-architectural attack involves recognizing some anomalous or malicious pattern of execution. The prevalent technique to classify attacks is to discover features that can provide distinct boundaries between these attacks and benign programs using some statistical method or Machine learning (ML) algorithms. Owing to this, detection-based techniques are more prudent at identifying the transmission of leakage, where the attacker performs distinct operations in the cache to glean the secrets.

A widely popular technique to capture program execution behavior is the use of Hardware Performance Counters (HPCs). These are registers provided by the hardware designer, to monitor certain micro-architectural events in the system. Originally intended for debugging purposes, over the last two decades, HPCs have been shown to profile programs to detect anomalies, malware [20] and specific micro-architectural attacks [3, 16, 35, 41, 42, 68], including those based on transient execution. Such solutions do not detect the anomalies in transient execution, but the step where the attacker gleans the sensitive data.

Another approach to using HPCs for attack detection is presented by the authors in [27]. It uses the observation that contention in a resource leaks information only when it is cyclic, meaning domain A interferes with domain B and sequentially domain B interferes with A. Thus the proposal to design a detection for such cyclic interference patterns using HPCs. While most detection techniques profile the attacker, there are approaches to profile the victim for anomalies [11]. The end goal of this design is to secure specific domains, rather than a blanket attack detection.

5 Conclusions

The last few years have seen several variants of transient micro-architectural attacks. The root cause in all these attacks is the unintended influence of speculatively executed operations with the hardware. Given the complexity of modern microprocessors, many new variants are likely to be discovered in the future. Next-generation microprocessors should be designed to not just prevent known attacks but should be resilient to future attacks as well. This would require security-aware design methodologies that involve the following.

- While there have been several countermeasures proposed, most have been evaluated in an ad-hoc manner. This makes it difficult to quantitatively compare countermeasures and gauge their effectiveness. There is an urgent need to standardize evaluation for security in microprocessors. These standards would provide methodologies to gauge the isolation between software entities. For example, a methodology that can quantify how well the OS is isolated from a userspace program. These methodologies could provide toolkits to analyze isolation or a suite of benchmark programs to evaluate the isolation.

- Pre and post-Silicon verification of hardware is mainly focused on functional aspects of the design. Automation tools are designed to minimize area, power, and boot performance. Security vulnerabilities, often fixed in hindsight, have proved expensive. Design automation tools should be augmented to validate for security early in the design phase. This can be a daunting task due to the vast state space of modern microprocessors. Artificial Intelligence (AI) is a promising tool that could help design automation for security. Although the use of AI in Electronic Design Automation (EDA) is in its infancy, AI is finding applications to reduce design verification time and achieve more optimized designs.

- Proposed preventive-countermeasures are designed to stymie specific variants of the attacks. For example, countermeasures for Meltdown are unable to protect against the newer MDS attacks. With multiple attack variants expected in the near future, defense solutions are always catching up with the attacks.

  Detection-based countermeasures, on the other hand, can easily adapt to new attacks. However, most detection countermeasures work from software, and are slow and inaccurate. CPU hardware can be augmented with attack sensors that can detect attacks at runtime with far better accuracy. These sensors should be generic enough to be configured for new attack variants.

  An alternate methodology is to use watchdogs, which monitor processor behavior to detect ongoing attacks. Programmable watchdogs have been proposed in [19], and can be extended for micro-architectural attacks.

References

[1] PaX: ASLR Documentation. https://pax.grsecurity.net/docs/aslr.txt, Accessed: 2021-3-2.

[2] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 132–144. IEEE, 2020.

[3] Manaar Alam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Victims can be saviors: A machine learning–based detection for micro-architectural side-channel attacks. *J. Emerg. Technol. Comput. Syst.*, 17(2), January 2021.

[4] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *28th International Conference on Parallel Architectures and*

Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019, pages 151–164. IEEE, 2019.

[5] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015.*, 2015.

[6] Daniel J. Bernstein. Cache-timing Attacks on AES, 2005.

[7] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 785–800. ACM, 2019.

[9] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[10] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita, editors, *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, pages 224–235. ACM, 2018.

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018.

[13] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 54–72. IEEE, 2020.

[14] 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. Fallout: Leaking data on meltdown-resistant cpus. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 769–784. ACM, 2019.

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. *IEEE Secur. Priv.*, 18(3):28–37, 2020.

[16] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.*, 49(C):1162–1174, December 2016.

[17] Intel Corporation. 11th Generation Intel Core Processor Desktop Datasheet, Volume 1, Revision 003, 2021. https://cdrdv2.intel.com/v1/dl/getContent/634648, Accessed: 2022-2-6.

[18] Intel Corporation. 12th Generation Intel Core Processor Desktop Datasheet, Volume 1, Revision 004, 2022. https://cdrdv2.intel.com/v1/dl/getContent/655258, Accessed: 2022-2-6.

[19] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. Phmon: A programmable hardware monitor and its security use cases. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 807–824. USENIX Association, 2020.

[20] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 559–570, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Abhijitt Dhavlle, Raj Mehta, Setareh Rafatirad, Houman Homayoun, and Sai Manoj Pudukotai Dinakarrao. Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks. In *21st International*

Symposium on Quality Electronic Design, ISQED 2020, Santa Clara, CA, USA, March 25-26, 2020, pages 161–166. IEEE, 2020.

[22] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity Mitigation of Cache Side-Channel Attacks. *TACO*, 8(4):35, 2012.

[23] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 61. ACM, 2019.

[24] Ed Younis Gonzȧlez Abraham, Ben Korpan and Jerry Zhao. Spectrum : Classifying , replicating and mitigating spectre attacks on a speculating risc-v microarchitecture. 2018. https://people.eecs.berkeley.edu/ kubitron/courses/cs262a-F18/projects/reports/project4_report.pdf, Accessed: 2021-4-4.

[25] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[26] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd M. Austin, and Mohit Tiwari. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 57–72. ACM, 2019.

[27] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd M. Austin, and Mohit Tiwari. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 57–72. ACM, 2019.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, page 191–205, USA, 2013. IEEE Computer Society.

[29] Intel. Intel C++ Compiler Classic Developer Guide and Reference. https://software.intel.com/content/dam/develop/external/documents/ cpp_compiler_classic.pdf, Accessed: 2021-3-2.

[30] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 974–987. IEEE Computer Society, 2018.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

[33] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In Christian Rossow and Yves Younan, editors, *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. USENIX Association, 2018.

[34] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using CFI informed speculation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 39–53. IEEE, 2020.

[35] Congmiao Li and Jean-Luc Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In Vladimir Getov, Jean-Luc Gaudiot, Nariyoshi Yamai, Stelvio Cimato, J. Morris Chang, Yuuichi Teranishi, Ji-Jiang Yang, Hong Va Leong, Hossain Shahriar, Michiharu Takemoto, Dave Towey, Hiroki Takakura, Atilla Elçi, Susumu Takeuchi, and Satish Puri, editors, *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, pages 588–597. IEEE, 2019.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[37] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 203–215. IEEE Computer Society, 2014.

[38] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[39] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2109–2122. ACM, 2018.

[40] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 118–129. IEEE Computer Society, 2012.

[41] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters. In Jakub Szefer, Weidong Shi, and Ruby B. Lee, editors, *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*, pages 1:1–1:8. ACM, 2018.

[42] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. WHISPER: A tool for run-time detection of side-channel attacks. *IEEE Access*, 8:83871–83900, 2020.

[43] Institute of Applied Information Processing and Communications (IAIK). Meltdown Proof-of-Concept. https://github.com/IAIK/meltdown, Accessed: 2021-3-2.

[44] Institute of Applied Information Processing and Communications (IAIK). ZombieLoad PoC. https://github.com/IAIK/ZombieLoad, Accessed: 2021-3-2.

[45] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive*, 2005:280, 2005.

[46] Colin Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan*, 2005.

[47] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 775–787. IEEE Computer Society, 2018.

[48] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 360–371. ACM, 2019.

[49] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021. Intel Bounty Reward.

[50] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.

[51] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 723–735, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Daniel Sánchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In Ravi Iyer, Qing Yang, and Antonio González, editors, *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 57–68. ACM, 2011.

[53] Matthias Schunter. Intel software guard extensions: Introduction and open research challenges. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, page 1, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Michael Schwarz, Moritz Lipp, Claudio Canella, R. Schilling, F. Kargl, and D. Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[55] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 753–768. ACM, 2019.

[56] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2019.

[57] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.*, pages 639–656, 2019.

[58] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 88–105. IEEE, 2019.

[59] Guanhua Wang, S. Chattopadhyay, Ivan Gotovchits, T. Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *ArXiv*, abs/1807.05843, 2018.

[60] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 74:1–74:6. ACM, 2016.

[61] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In Dean M. Tullsen and Brad Calder, editors, *ISCA*, pages 494–505. ACM, 2007.

[62] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[64] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 675–692. USENIX Association, 2019.

[65] You Wu and Xuehai Qian. Reversispec: Reversible coherence protocol for defending transient attacks. *CoRR*, abs/2006.16535, 2020.

[66] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *22nd Symposium on Reliable Distributed*

Systems (SRDS 2003), 6-8 October 2003, Florence, Italy, page 260. IEEE Computer Society, 2003.

[67] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, page 1076. ACM, 2019.

[68] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquín García-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, volume 9854 of *Lecture Notes in Computer Science*, pages 118–140. Springer, 2016.

[69] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 1138–1152. IEEE, 2020.