Speculative Load Forwarding Attack on Modern Processors

Hasini Witharana and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA

ABSTRACT

Modern processors deliver high performance by utilizing advanced features such as out-of-order execution, branch prediction, speculative execution, and sophisticated buffer management. Unfortunately, these techniques have introduced diverse vulnerabilities including Spectre, Meltdown, and microarchitectural data sampling (MDS). Although Spectre and Meltdown can leak data via memory side channels, MDS has shown to leak data from the CPU internal buffers in Intel architectures. AMD has reported that its processors are not vulnerable to MDS/Meltdown type attacks. In this paper, we present a Meltdown/MDS type of attack to leak data from the load queue in AMD Zen family architectures. To the best of our knowledge, our approach is the first attempt in developing an attack on AMD architectures using speculative load forwarding to leak data through the load queue. Experimental evaluation demonstrates that our proposed attack is successful on multiple machines with AMD processors. We also explore a lightweight mitigation to defend against speculative load forwarding attack on modern processors.

1 INTRODUCTION

Modern processors provide high performance by using different techniques such as out-of-order execution, speculative execution and utilizing microarchitectural structures (internal buffers). Out-of-order execution utilizes execution units of a CPU as much as possible allowing the CPU to execute instructions speculatively. Instead of executing all the instructions in sequential order, the processor executes the instructions as early as possible. Utilization of internal buffers such as caches and load/store buffers have reduced the latency of fetching data always from the main memory. Even though speculative and out-of-order execution are valuable techniques to obtain high performance for modern processors, they have also introduced serious security vulnerabilities.

Out-of-order and speculative behavior often leads the instructions to be executed out-of-turn, earlier than their expected in-order execution. Transient execution is when instructions executed out-of-order and erroneously changing the microarchitectural states of a CPU. The reasons for transient execution may be a delay in exception handling, misprediction or micro-code assisted event. The result of a transient execution may not be architecturally visible, but it can change the microarchitectural states such as cache, store buffers, load buffers, etc. Attackers can gain advantage of these microarchitectural changes and leak valuable information using CPU side-channel attacks such as Spectre [12, 13], Meltdown [14], Foreshadow [21, 25] and microarchitecture data sampling (MDS) [6, 18, 19, 22, 23]. Due to the high security risk imposed by these attacks, modern processors have developed several mitigation techniques.

All major operating systems have implemented countermeasures (KAISER [9], [17]) against transient execution attacks that can introduce significant performance loss. Most systems encourage programmers to use serialization such as LFENCE [7], when needed to avoid transient execution. Intel CPUs have been identified as vulnerable for most of the transient execution attacks. Therefore, Intel has introduced new silicon with hardware mitigation to prevent Meltdown type transient attacks while maintaining the expected high performance. Unlike Intel processors, AMD processors are believed to be resistant against Meltdown/MDS type attacks [2].

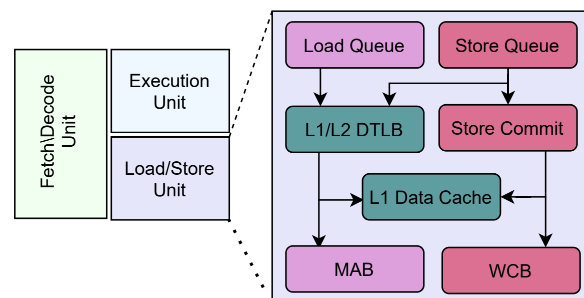

Figure 1: AMD Zen family architecture

In this paper, we present a Meltdown/MDS type attack on AMD processors to leak data from the load queue that stores the previous loads to hide the latency of loading value from memory. Specifically, this paper makes the following major contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549417

- We have identified a speculative load forwarding vulnerability in AMD Zen2 and Zen+ architectures.
- We present a Meltdown type attack that can lead to information leakage through load queue in the same user space.
- Experimental evaluation shows that our proposed attack is successful on multiple machines with AMD processors.
- We explore potential lightweight mitigation methods to defend against speculative load forwarding attacks.

The remainder of the paper is organized as follows. Section 2 discusses the background and related efforts. Section 3 presents an overview of the proposed speculative load forwarding attack. Section 4 describes our attack on AMD Zen family architectures. Section 5 explores mitigation techniques. Section 6 presents the experimental results. Finally, Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

We first provide relevant background on AMD Zen family architecture and out-of-order execution. Next, we survey related work on transient execution attacks as well as cache timing attacks.

2.1 AMD Architecture

Figure 1 shows a high-level overview of AMD Zen family. AMD Zen family microarchitecture contains three main stages: Fetch/Decode unit, Execution unit, and Load/Store unit. The Fetch/Decode unit fetches the instructions, decodes, and issues the μ-operations to the Execution unit. Execution unit executes the instruction out-of-order while loading and storing values through the Load/Store unit. As shown in Figure 1, Load/Store unit consists of Load Queue, Store Queue, L1 data cache, and Translation Lookaside Buffer (TLB). **TLB** is a memory cache that stores the virtual to physical address mapping of recently accessed addresses. TLB reduces the access time of a memory location. According to [5], a TLB hit is required for load speculation. **Load Queue** stores the previous loads to hide the latency of loading value from the main memory. Load Queue allows speculative load forwarding mechanism if there is a TLB hit for the address.

2.2 Out-of-Order Execution

Modern processors implement out-of-order execution similar to Tomasulo [20, 24] algorithm. Tomasulo algorithm consists of a unified reservation unit commonly called as Reorder Buffer (ROB), which preserve the original programming order and stores the executed data internally without changing any architectural states (actual registers). Out-of-order execution units typically consist of three stages: (1) in-order register allocation and renaming, (2) out-of-order instruction execution, and (3) in-order retirement of instructions. In-order register allocation unit eliminates the hazards such as Write-after-Read (WAR) and Write-after-Write (WAW) by register renaming, and sends the μ-operations to the ROB. Out-of-order execution unit eliminates Read-after Write (RAW) hazards by stalling the μ-operations, and issues the instructions to execution units when all the operands are available. When the execution unit returns the results of a μ-operation, ROB and internal buffers store the data and mark the instruction as completed. In-order retirement unit retires instructions in-order irrespective of the order the instructions actually got executed. Instructions commit in the original program order, and the data values are written to the actual registers. In the retirement stage if there is a transient execution, ROB detects that retirement should not happen. Hence all the executed instructions after the transient execution in the ROB will get flushed and the processor will compute the instructions again without speculation.

2.3 Related Work

A vast majority of architectural attacks consist of two phases. The actual attack is performed (leak is introduced) in the first phase. The second phase performs the leak detection using CPU cache side-channel attacks. We briefly describe these two phases.

Transient Execution Attacks: Most of the microarchitectural features trigger security violations in modern processors leaking victims' secret through transient execution. There are several types of attacks such as Spectre, Meltdown and microarchitecture data sampling (MDS). Spectre [12] exploits the speculative behaviour of branch prediction to read arbitrary memory from the victims process. Meltdown [14] exploits out-of-order execution and side-channel attacks on processors to read kernel memory from unprivileged user space. MDS [6, 19, 23] is a Meltdown type attack to leak through internal buffers. MDS can leak in-flight data from CPU internal buffers such as store buffer, line-fill buffers, and load ports buffer. Table 1 shows a comprehensive comparison of existing transient attacks with respect to different architectures (Intel and AMD). All Intel and AMD processors are vulnerable to Spectre attacks. Based on published results, only Intel CPUs are vulnerable for Meltdown and MDS attack. There have been prior efforts on conducting Meltdown/MDS type attacks on AMD CPUs. The authors in [16] have introduced an MDS type attack on AMD machines exploiting the store queue when the secret is also presented in L1 cache. Our proposed attack is different from [16] since we leak through the load queue even when the secret is not in the cache.

Even though AMD and Intel are x86 based systems, their architectures are different from each other [8]. AMD architecture is less vulnerable to MDS/Meltdown type attacks, compared to the Intel CPU. The key to classical meltdown is that

Table 1: Evaluating different transient attacks on Intel and AMD architectures.

	Spectre	Meltdown	Store Buffer Related Attacks	Fill Buffer Related Attacks	Load Buffer Related Attacks
Intel	[12]	[14]	[6, 16]	[19, 23]	[23], Our Approach
AMD	[12]		[16]		Our Approach

Intel CPUs don't flush under-privileged TLB hits. In contrast to Intel, AMD's load execution units / TLB are designed differently, with privilege checking for loads applied earlier or differently. This makes the AMD architecture less vulnerable to Meltdown. MDS tries to leak data through micro-architectural structures. For Intel architectures, different techniques such as microcode assists, simultaneous multithreading and TSX faults are used for MDS attacks. However, AMD architecture handles such errors differently making the MDS attack much harder than Intel machines. Some of the techniques that prevent MDS at AMD is the TLB flush across kernel and userspace.

Cache Timing Attacks: To reduce the memory latency, the CPUs utilize memory buffers (caches) to store data based on spatial and temporal nature of accessed data. There are typically multiple levels of caches. Primary cache (e.g., L1) is very fast but the cache size is small, whereas secondary cache (e.g., L2 and L3) is relatively slower but more spacious than the primary cache. The difference of access cycles from cache and main memory has lead to CPU cache timing attacks [15]. By timing the access time of a data, it is easy to recognize whether the data is coming from the cache or the main memory. There are several types of cache timing attacks such as FLUSH+RELOAD and PRIME+PROBE. In FLUSH+RELOAD [10, 26], an attacker first flushes the cache to evict the data already in the cache. Next, the attacker reloads the data and calculates the access time for a cache hit to identify whether any of the data has been accessed by the victim. This does not work when the critical data is not shared. In PRIME+PROBE [11], an attacker populates all cache ways rather than flushing the cache. The attacker will calculate the access time to identify a cache miss, which ensures that the victim has accessed that data. In this paper, we use FLUSH+RELOAD attack to identify the leak. *To the best of our knowledge, there are no prior efforts in leaking data using load queue from AMD architectures.*

3 OVERVIEW

The overview of our proposed speculative load forwarding attack is shown in the Figure 2. Our attack involves five steps: S_1, S_2, S_3, S_4, and S_5. The first step is initiated by a victim. The victim has an $address_1$ which has the value *secret*. The victim loads $address_1$ and this load information is populated in the load queue. The goal of the attacker is to steal the *secret* value loaded by the victim. The second step is initiated by the attacker. The attacker first flushes the cache. Next, the attacker tries to get the value *secret* by loading the $address_2$ in the third step. The load of *address*2 will cause a transient execution and the value of $address_1$ (*secret*) will be passed to the $L1$ cache in the fourth step. Finally, the attacker performs a CPU cache side-channel attack to identify the value of $address_1$. In the final step, the attacker reloads the entries from the cache to identify the *secret* leaked through the load queue. Section 4 provides detailed attack describing how to transiently forward the value of $address_1$ via $address_2$ load.

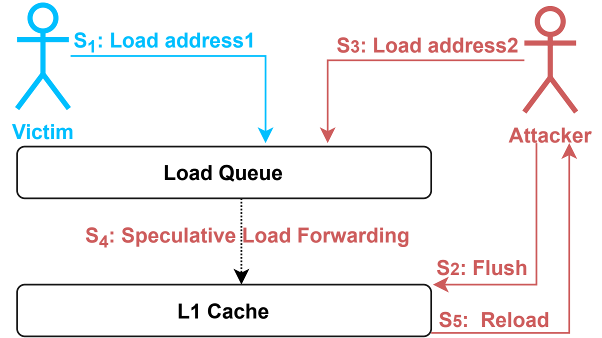

Figure 2: Overview of speculative load forwarding attack

4 LOAD FORWARDING ATTACK

Listing 1: Overall attack

```
1.  Victim_Load();
2.  Initialize();
3.  for (int i = 0; i < Rounds; ++i) {
4.     for (int j = 0; j < Experiments; ++j) {
5.         Flush();
6.         Attack_Load();
7.         Reload();
8.     }
9.    Local_Wins();
10. }
11. Winner();
```

In this section, we present our speculative load forwarding attack. Listing 1 shows the overall attack. As shown in line 1, in the victim space, the *secret* is stored in $address_1$ and the victim loads this address. Next, the attacker generates the $address_2$ and initializes required buffers in line 2. Lines 3-10 shows the outer 'for' loop to iterate through the rounds. Lines 4-8 shows the inner for loop for conducting several number of experiments for each round. For each experiment, the attacker performs three tasks. The first task is flushing the cache (line 5). The second task is loading $address_2$ (line 6). The third task is reloading values from the cache (line 7). After running all the experiments, the leak detection is executed and a local winner is identified in line 9. This process is continued for

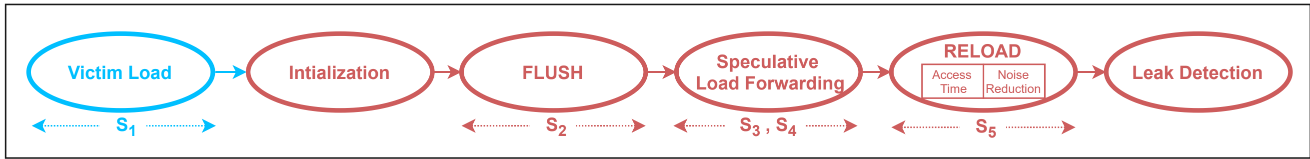

Figure 3: Attack narrative of speculative load forwarding

all the rounds. Based on the local winners of each round, the final winning candidate is identified in line 11.

Figure 3 shows a simplified flow diagram of our overall attack that consists of six major steps: victim load, initialization, flush, speculative load forwarding, reload and leak detection. The remainder of this section describe these steps in detail.

4.1 Victim Load

In this attack primitive, victim space has an $address_1$ which has the value *secret*. As shown in Listing 2, victim has allocated memory for $address_1$ and then *memset* the value *secret* for the address. This is equivalent to storing the value *secret* in $address_1$. The victim loads the $address_1$ as shown in the line 3. This load will populate an entry in the load queue. As shown in line 4, we flush $address_1$ to make sure that the secret value is not in the cache. This also guarantees that the load value does not have to be present in the cache for this attack to be successful.

Listing 2: Victim_Load()

```
/* Initialize address1 with value secret. */
1. char *address1 = malloc(20 * PG_SIZE);
2. memset(address1, secret, 20 * PG_SIZE);

/* Victim loads address1. */
3. int val = addr1[offset];

/* Flush address1. */
4. flush(&addr[offset]);
```

4.2 Initialization

In the attacker space, attacker needs to generate $address_2$ with last 48 bits of the address matching the last 48 bits of the $address_1$. This bit-level matching of the two addresses are important because only the last 48 bits are checked through the TLB and load queue in AMD architectures. The loading of $address_2$ should trigger an exception, generating a window to load the secret speculatively. We are using non-canonical address violation as this exception [3]. A non-canonical address means the upper bits from 49 to 64 should not be all 0's or 1's. If all the upper bits are neither 1 or 0, the access of this address create a non-canonical address violation. Since the non-canonical violation is detected later in the instruction pipeline, it provides a window of opportunity to load the secret speculatively creating a transient execution. We generate the $address_2$ fulfilling the requirements as shown in Listing 3 (line 1). Listing 4 shows illustrative examples of $address_1$ and $address_2$.

Listing 3: Initialize()

```
/* Initialize address2 with last 48 bits
matching address1 and non canonical error. */
1. char *address2 = address1|0xff00000000000000;

/* Initialize buffers */
2. char *cacheAccessArray = malloc(256*PG_SIZE);
3. uint64_t numberOfWins[256];
4. uint64_t totalAccessTime[256];
5. memset(numberOfWins, 0x0, 256);

/* Initialize Exception Handler */
6. Catch_Segmentation_Fault();
```

As shown in Listing 3 (line 2), we allocate a buffer *cacheAccess Array* to load the secret to the cache. To calculate the number of wins for all the rounds, *numberOfWins* buffer is allocated and initialized in line 3 and line 5, respectively. In order to calculate the total access time of buffer elements, *totalAccessTime* buffer is initialized in line 4. Since the $address_2$ has the non-canonical violation error, accessing this address should throw an exception. When the execution pipeline identifies the exception, it will terminate the program. We avoid the program termination by catching the exception as shown in line 6.

Listing 4: Example of $address_1$ and $address_2$

```
address1 - 0x0000562419f9a000
address2 - 0xff00562419f9a000
```

4.3 FLUSH

In the attack narrative, we are using FLUSH+RELOAD as the cache timing attack to identify the secret. The first step of FLUSH+RELOAD is to flush the cache so that all the values already in the cache will get evicted. The *cacheAccessarray*, which is initialized and allocated in Listing 3, is used for the FLUSH+RELOAD. As shown in the Listing 5, we first flush the cache to remove any cache line with *cacheAccessarray*.

Listing 5: Flush()

```
/* Flush buffer entries. */
1. for (int i=0; i<256; i++) {
2.     cacheAccessArray[i*PG_SIZE] = 0;
3.     flush(&cacheAccessArray[i*PG_SIZE]);
4. }
```

4.4 Speculative Load Forwarding

In this step, the attacker tries to load the $address_2$ as shown in the Listing 6. For this load, only the last 48 bits will be checked in the TLB and the load queue. First, there will be a TLB hit because the $address_1$ entry is already in the TLB. Next, the $address_2$ load will speculatively get the value of $address_1$ from the load queue with the last 48-bit match. Since the non-canonical violation is not detected at the time of the 48-bit match, secret will be speculatively forwarded through $address_2$.

Note that the *secret* value is not yet disclosed to the cache. To get the secret value in the cache, *cacheAccessArray* is used. From the transient execution *cacheAccessArray* gets the *secret* as the computation of $address_2[offset]$ and that buffer ($cacheAccessArray[PG_SIZE * secret]$) element will be loaded into the cache. When the non-canonical violation is detected, all the execution will get reverted. However, the *secret* will be already in the cache through the *cacheAccessArray*. Once the exception is detected, the program will throw the exception and our exception handler avoids the termination of the program.

Listing 6: Attack_Load()

```
/* Attacker loads address2. */
cacheAccessArray[PG_SIZE * addr2[offset]];
```

4.5 RELOAD

Once the secret is loaded to the cache through the *cacheAccess array* buffer, we perform the second step of FLUSH+RELOAD attack to identify the secret by analyzing the access time. First, we discuss access time analysis. Next, we describe several noise reduction techniques.

Access Time: As shown in the Listing 7 (lines 1 - 6), the access time for all the 256 buffer entries are calculated. If the buffer entry is already in the cache, access time will be fast. However, if the buffer entry is not in the cache, this entry has to be fetched from the main memory and it will lead to significantly longer access time compared to accessing an entry in the cache. This drastic time difference to access from cache and main memory is used to identify the secret value speculated through the attacker load. If the secret value is successfully speculated and loaded to the cache, the access time of the secret value should be faster than the other buffer elements.

Listing 7: Reload()

```
/* Calculate the time access of each entry in
buffer*/
1. for (int i=0; i<256; i++) {
2.     t1 = rdtscp();
3.     *(cacheAccessArray + PG_SIZE * i);
4.     t2 = rdtscp() - t1;
5.     totalAccessTime[i] += t2
6. }
```

Noise Reduction Techniques : In FLUSH+Reload attacks, prefetching introduces noise, since some of the buffer elements can be loaded to cache through prefetching. To reduce the latency of accessing frequently used memory locations, CPUs typically use software/hardware prefetchers. Prefetchers preload instruction and data into cache based on the current data being loaded. Software prefetchers improve the performance by automatically inserting loads through the compiler for instances such as for loops.

When performing FLUSH+RELOAD, it will be hard to identify the secret due to the noise. The noise will be present because the prefetcher will try to add values to the cache based on the spatial locality. When looping through the buffer, prefetcher will try to load the nearby entries of the buffer understanding the pattern of accessing value increasing one by one or simply by the compiler for the loop. This will increase the noise in FLUSH+RELOAD attack. We can avoid prefetching by removing the access pattern of the buffer. So we used a simple hash which will load all the 256 entries but not in the increasing order of one. This is shown in line 1 of the Listing 8.

Listing 8: Avoiding prefetcher

```
for (int i=0; i<256; i++) {
/* A simple hash to prevent prefetching */
1.     int hash = ((j *167)+13 & (0xff));
2.     t1 = rdtscp();
3.     *(cacheAccessArray + PG_SIZE * hash);
4.     t2 = rdtscp() - t1;
}
```

4.6 Leak Detection

Once the secret is in the cache, we can identify the leak by calculating the access time for each entry in the *cacheAccessArray*. There are several ways to identify a leak. One way is using a threshold and checking whether the access time is less than the threshold. If the access time of an entry of the buffer is less than the threshold that entry can be considered as coming from the cache. This threshold value can vary based on the processors as well as the system specification. If using a threshold as an identifier, attacker first need to understand the best threshold to use depending on the scenario.

Listing 9: Local_Wins()

```
/* Calculate minimum average access time */
1. uint64_t min, avgAccessTime = 0;
2. int localWinner = -1;
3. for (int j = 0; j < 256; ++j) {
4.   avgAccessTime=totalAccessTime[j]/Experiments
5.   if (avgAccessTime <= min) {
6.       min = avgAccessTime;
7.       localWinner = j;
8.   }
9. }
10. ++numberOfWins[localWinner];
```

There are ways to identify the secret without using a threshold. One way is to calculate the average access time for all the buffer elements and to find the minimum average time. The buffer element with minimum average time can be considered as the secret. In our attack, we are calculating the minimum average access time to identify the leak. As shown in Listing 9 (line 4), average access time is calculated for each entry in the *cacheAccessArray* by dividing the *totalAccessTime* by number of experiments. If the calculated average access time is less than or equal to the minimum average access time, that buffer entry will be marked as a local winner (line 7).

An attacker should ensure that the leak is accurate and consistent - an attack should always return the secret irrespective of the noise. To guarantee that the leak is consistent, the attack should be performed for different number of rounds and check how many number of rounds the secret can be identified. After the attack is conducted for all the rounds, winner is calculated as shown in Listing 10 (lines 3 - 7). The *cacheAccessArray* buffer element with the highest number of wins from all the rounds will be selected as the winner (secret).

Listing 10: Winner()

```
/* Calculate winner */.
1. int winner;
2. int64_t max = -1;
3. for(int i = 0; i < 256; ++i){
4.     if(numberOfWins[i] > max) {
5.         max = numberOfWins[i];
6.         winner = i;
7.     }
8.}
```

5 MITIGATION METHODS

In this section, we explore different mitigation techniques for our speculative load forwarding attack on AMD Zen family architectures. We have reported the vulnerability to AMD and they suggested that the LFENCE [4] can be used as a mitigation against our proposed attack. The LFENCE instruction serializes load instructions, such that the speculative execution will stop until the load instruction is committed. The ideal mitigation for the attack should reside in the victim space after the victim load. However, the use of LFENCE immediately after the victim load (as shown in Listing 11) does not mitigate the attack.

Listing 11: LFENCE mitigation after victim load

```
/* Victim Load */
1. int val1 = address1[offset];
2. LFENCE;
/* Attacker Load */
3. cacheAccessArray[PG_SIZE * address2[offset]];
```

In the Listing 11, line 3 can be broken down into two pieces, a load from $address_2$ and a (transient) load from *cacheAccess Array*. If an LFENCE is added between those two loads (as shown in Listing 12), it would allow the processor to recognize the non-canonical address violation before the cache line from cacheAccessArray is loaded.

The mitigation of using LFENCE is only working on the attacker space. This is unrealistic since we cannot expect an attacker to use LFENCE while writing the attack code. An actual mitigation should reside in the victim space where the victim should be able to protect valuable information by preventing speculation. For MDS attacks, Intel has introduced VERW mitigation which will clear the internal buffers preventing any speculation on loads or stores [1]. The VERW mitigation can be applied on the victim space as well. We suggest a mitigation similar to VERW should be implemented by AMD if the mitigation to be applied in the victim space.

Listing 12: LFENCE mitigation after attacker load

```
/* Victim Load */
1. int val1 = address1[offset];
/* Attacker Load */
2. int val2 = address2[offset];
3. LFENCE;
4. cacheAccessArray[PG_SIZE * val];
```

6 EXPERIMENTS

This section demonstrates the effectiveness of the proposed attack. First, we describe our experimental setup. Next, we present the results for the attack including noise reduction and attack accuracy. Finally, we discuss the mitigation results.

6.1 Experimental Setup

Our experimental setup consists of different AMD Zen family machines as shown in Table 2. The attack code outlined in Listing 1 is written using C and compiled using zero optimization with gcc. In the experimental setup, the victim loads the value 0x88 (decimal value 136) and the attacker tries to leak the value 0x88. Table 2 provides the details of the machines where we are able to reproduce the leak. These results elaborate that this vulnerability is exploitable in Zen2 as well as Zen+ architectures.

Table 2: Details on machines/microarchitectures

CPU	Year	Microcode	μ Architecture
AMD Ryzen 5 PRO 1600	2017	0x8001137	Zen2
AMD Ryzen 5 3500U	2019	0x8108109	Zen+
AMD Ryzen 5 3600U	2019	0x8701021	Zen+

6.2 Attack Results

We have verified that we can leak information through the load queue in several AMD machines. In the attack we conducted, victim loads some secret. Attacker speculatively leaks

this value in the same user space, then use FLUSH+RELOAD techniques to expose the secret value as described in Section 4. We have created a covert channel and was able to leak 100 bytes/s. For the rest of the results in this section, we only consider the leak of one byte (0x88).

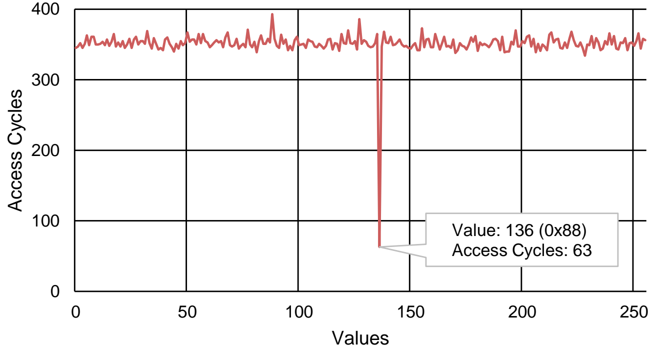

Figure 4: Average access time with noise cancellation

As described in the Section 4, we used noise reduction techniques to achieve high accuracy in the leak. Figures 4 and 5 show the average access time (cycles) of the 256 elements in the *cacheAccessArray* after running the attack for five rounds and 100,000 experiments per round, with and without noise cancellation respectively. Figure 4 (with noise cancellation) illustrates that only for value 0x88, the access time is drastically small compared to the other values in the buffer. Therefore, it is easy to recognize the leak. According to Figure 5 (without noise cancellation), the access time of most of the buffer elements are low since most of the values are prefetched to the cache by the prefetcher. These results illustrate that the use of noise reduction techniques in transient execution attacks is vital to improve the accuracy of the attack.

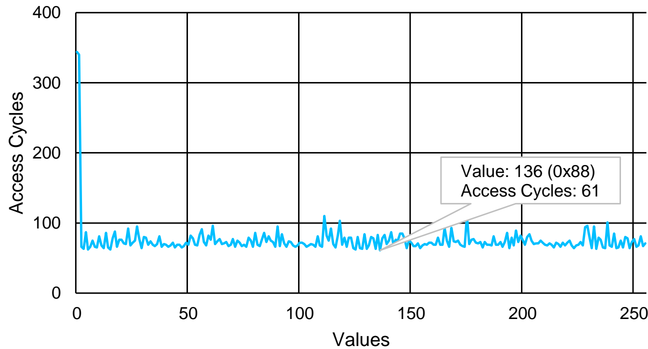

Figure 5: Average access time without noise cancellation

We found that even without noise cancellation techniques, we can achieve improved accuracy by increasing the number of rounds and number of experiments. Table 3 and Table 4 show the number of rounds (first column), number of experiments per round (second column), average access time (cycles), and number of wins per rounds of the secret value with and without noise cancellation. The number of rounds and experiments in the two tables corresponds to the 'Rounds' (line 3) and 'Experiments' (line 4) in the Listing 1. One win means that for one round, the secret value has the lowest average access time and the secret is identified successfully.

Table 3: Average access time of secret and number of wins per round with fixed number of experiments (100) and increasing number of rounds.

Rounds	Experiments	Without Noise		With Noise	
		Acc.Time	Wins	Acc.Time	Wins
1	100	116	1/1	116	0/1
2	100	85	2/2	78	0/2
3	100	92	3/3	71	0/3
4	100	69	4/4	70	0/4
5	100	69	5/5	84	1/5
6	100	64	6/6	64	1/6

In the Table 3, we increase the number of rounds while maintaining a fixed number of experiments (100), and observe the number of wins per round of the secret value with and without noise cancellation. With noise cancellation, even with one round and 100 experiments the leak is identified. However, without noise cancellation, it is hard to guarantee the leak with 100% accuracy even with 6 rounds and 100 experiments.

Table 4: Average access time of secret and number of wins per round with fixed number of rounds (5) and increasing number of experiments.

Rounds	Experiments	Without Noise		With Noise	
		Acc.Time	Wins	Acc.Time	Wins
5	10	83	5/5	140	0/5
5	100	95	5/5	65	1/5
5	1000	96	5/5	76	1/5
5	10000	67	5/5	74	2/5
5	100000	63	5/5	61	5/5

In the Table 4, we increase the number of experiments while maintaining a fixed number of rounds (5), and observe the number of wins per round of the secret value with and without noise cancellation. With noise cancellation, the secret is always identified with 100% accuracy even with 10 experiments. When we increase the number of experiments to 100,000, we are able to identify the leak successfully for all five rounds even without noise cancellation. This is because for all the experiments the secret will have a lower access time but the access time of the other values will depend on the behavior of the prefetcher. This shows that by using a large number of experiments we can get accurate results even without noise cancellation.

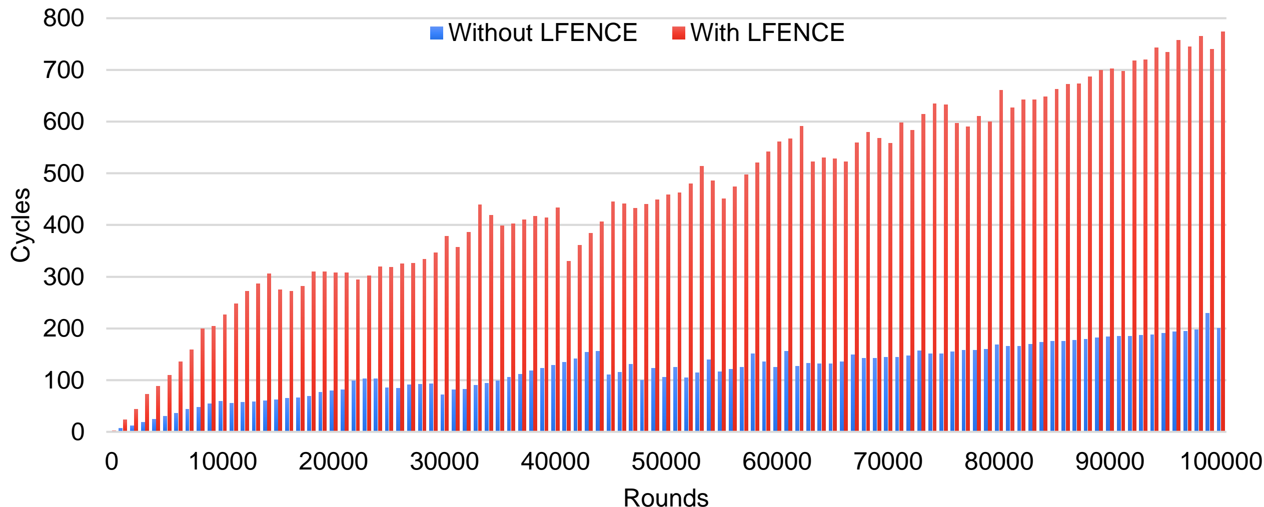

Figure 6: Average time taken to execute load followed by load with and without LFENCE between them.

6.3 Mitigation Results

As discussed in Section 5, we explored two mitigation techniques: (1) LFENCE after victim load, and (2) LFENCE after attacker load. Figure 7 shows the average access time for the buffer values with respect to the two mitigation techniques. Clearly, LFENCE after victim load (blue) does not stop the leak. We can observe that the secret value (0x88) has the lowest average access time. The LFENCE after attacker load (red) stops the leak. This is expected because using LFENCE after attacker load serializes the loads and stops the speculation.

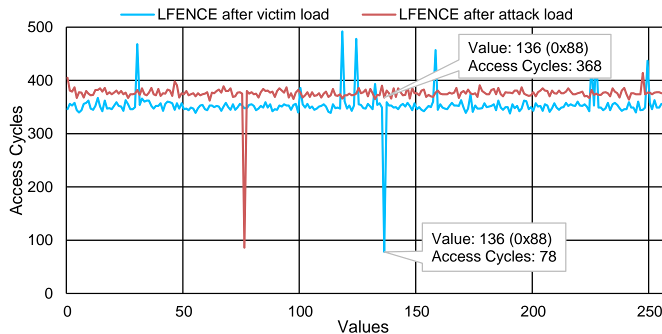

Figure 7: Average access time with LFENCE mitigation

To evaluate the performance impact of applying LFENCE between loads, we conducted an experiment on AMD machine (AMD Ryzen 5 3500U) and the results are shown in Figure 6. We calculated the average time (cycles) taken to execute load followed by load with and without LFENCE in between loads, while increasing the number of loads using rounds. As shown in the figure, LFENCE between two loads increases the execution time compared to normal execution. According to the experiment, the average performance impact is 2.86 times. Note that overall performance impact would be less since approximately 20% of all instructions are load in typical programs. In other words, assuming 20% load instructions in a program, the overall performance penalty would be approximately 50%. Either way, using LFENCE between all the loads to prevent our attack is not feasible due to the performance impact. Therefore, the insertion of LFENCE must be done appropriately with manual supervision by identifying most vulnerable load forwarding scenarios.

7 CONCLUSION

Modern processors utilize out-of-order and speculative execution to deliver high performance. However, these advanced features have made the systems vulnerable to transient execution attacks and CPU side-channel attacks. Most of the security analysis studies are carried out on Intel processors, and there is a need for similar studies on other processors such as AMD and ARM. While Intel processors are vulnerable to many transient execution attacks, AMD is believed to be resistant against Meltdown/MDS type attacks. In this paper, we challenge that belief and present a Meltdown/MDS type attack on AMD Zen family architectures. Our experimental results demonstrate that a speculative load forwarding attack through the load queue is possible for Zen architectures. We also evaluate different mitigation techniques to defend against speculative load forwarding attacks.

ACKNOWLEDGMENTS

This work was partially supported by the grant from Semiconductor Research Corporation (2020-CT-2934).

8 DISCLOSURE

We submitted proof-of-concept (PoC) exploit for the MDS/Meltdown type attack through load buffer to AMD on October 26, 2021. AMD has acknowledged us that this attack can be categorized to attack class disclosed in [3] on October 27, 2021.

REFERENCES

[1] 2019. Intel, "Microarchitectural Data Sampling / CVE-2018-12126,CVE-2018- 12127,CVE-2018-12130,CVE-2019-11091 / INTEL-SA-00233," 2019.

[2] 2021. AMD Product Security. https://www.amd.com/en/corporate/product-security.

[3] 2021. Transient Execution of Non-canonical Accesses. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1010.

[4] 2021. White Paper: Software Techniques for Managing Speculation on AMD Processors. https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf.

[5] 2021. White Paper: Speculation Behavior in AMD Micro Architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf.

[6] Claudio Canella et al. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*. ACM.

[7] Xing Fang, Jaejin Lee, and Samuel P Midkiff. 2003. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*. 285–294.

[8] Agner Fog. 2012. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering* 2 (2012).

[9] Daniel Gruss et al. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.

[10] David Gullasch et al. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *S&P*. IEEE, 490–505.

[11] Gorka Irazoqui et al. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *S&P*. IEEE.

[12] Paul Kocher et al. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[13] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[14] Moritz Lipp et al. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[15] 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.

[16] Saidgani Musaev and Christof Fetzer. 2021. Transient Execution of Non-Canonical Accesses. *arXiv preprint arXiv:2108.10771* (2021).

[17] Zhixin Pan and Prabhat Mishra. 2021. Automated detection of spectre and meltdown attacks using explainable machine learning. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 24–34.

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

[19] Michael Schwarz et al. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. 753–768.

[20] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.

[21] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient {Out-of-Order} Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[22] 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 *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[23] Stephan van Schaik et al. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[24] Chao Wang, Xi Li, Junneng Zhang, Xuehai Zhou, and Xiaoning Nie. 2013. MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 2 (2013), 1–26.

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

[26] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*. 719–732.