Optimization and Amplification of Cache Side Channel Signals

David A. Kaplan
Advanced Micro Devices, Inc.
Email: david.kaplan@amd.com

***Abstract*—In cache-based side channel attacks, an attacker infers information about the victim based on the presence, or lack thereof, of one or more cachelines. Determining a cacheline's presence, which we refer to as "reading the signal", typically requires testing the access time of the line using a suitably high precision timer. In this paper we introduce novel gadgets which leverage CPU speculation to enable modification of these signals, before they are read, for a variety of purposes. First, these gadgets enable an attacker to optimize cache-based side channel attacks by evaluating arbitrary logic functions on cacheline signals prior to their measurement. Second, we demonstrate amplification techniques that enable an attacker to read a signal even if no high precision timer is available. Combined, these techniques can be used to improve existing side channel attacks even if timer access is limited. We evaluate the effectiveness of these techniques on a modern x86 CPU and demonstrate that when properly tuned, cache side channel signals can be reliably modified with near 100% accuracy and are able to be read with a timer as coarse as 100ms or more.**

I. Introduction

The CPU cache has been widely used for side channels attacks in a variety of research. Side channel attacks have previously been demonstrated in the recovery of sensitive data (including AES[1], [2], [3], [4] and RSA[5] keys) in vulnerable cryptographic libraries. In 2018, Google Project Zero[6] and independent researchers[7] demonstrated a variety of speculation-based side channel attacks that leveraged the CPU cache to leak information from one program or protection domain to another. As the cache is a shared resource, side channel attacks have been demonstrated at the L1, L2 and L3[8], [9] levels on modern CPUs.

A variety of techniques for cache-based side channel attacks have been studied, including PRIME+PROBE[10], FLUSH+RELOAD[11], and others[12]. Regardless of the specific technique used, the attack typically works in 3 stages. First, the attacker puts the system into a desired state, perhaps by loading or evicting certain cache lines. Second, the victim is invoked resulting in some manipulation of the cache state. Finally, the attacker attempts to detect the manipulation of the cache state, thereby inferring information about the victim.

This paper focuses solely on the final stage of these attacks, which we call signal recovery. In the most basic FLUSH+RELOAD attack, a single cacheline may or may not be present, which we define as a signal of 1 or 0 respectively. The attacker typically reads the signal value by timing the access to this cacheline using the x86 RDTSC instruction, the `performance.now` JavaScript function, or another suitably high precision timer. The attack is often repeated many times in order to read more information and eliminate potential noise.

There are a variety of practical challenges an attacker faces when attempting to do signal recovery on modern systems. In response to Spectre, most web browsers have taken steps to dramatically reduce the precision of `performance.now` and eliminate user-created high precision timers created via SharedArrayBuffer. Microsoft[13] and Mozilla[14] initially reduced the resolution of `performance.now` to 20us with variable jitter of an additional 20us. WebKit[15] further reduced the resolution of `performance.now` to 1ms. As the time difference between a single cache hit and a miss is typically less than 100ns, this creates significant signal recovery challenges for any type of cache-based side channel attack.

Even if high precision timers are available to the attacker, signal recovery optimization may still be desired. Mitigations have been proposed[16], [17] to look for abnormally high numbers of calls to high precision timers (like RDTSC) to try to detect an in-progress side channel attack. In this paper we demonstrate ways to manipulate cache side channel signals *without reading them* thereby reducing the number of timing checks eventually needed. Besides avoiding detection, this may also lead to optimized attack implementations.

To our knowledge, this is the first paper to focus exclusively on this signal recovery aspect of cache side channels. Our specific contributions include:

- Introducing a generic primitive that operates on cache side channel signals without requiring a time source
- Introducing gadgets which use this generic primitive to perform logical operations on cache side channel signals on x86 platforms
- Demonstrating signal amplification techniques which allows for cache signal recovery with a timer granularity of over 100ms

The rest of the paper is organized as follows. Section II discusses the underlying hardware and related work in this space. Section IV introduces the generic primitive we use for signal modification with specific gadgets discussed in

Section V. Section VI and VII discuss the theory behind signal amplification and applications of signal modification. And Section VIII demonstrates the gadgets and algorithms on a modern x86 system. At the end of the paper, we briefly discuss potential mitigations and opportunities for further research.

Note: During the development of this paper, we became aware of similar research being conducted independently by another research team [18]. We wish to acknowledge this concurrent work, and thank the authors for their coordination and discussions around this topic. While the publication of the work by both teams were coordinated, the content and results presented in this paper represent the sole work of the listed author.

II. Background And Related Work

A. Cache Side Channels

Whenever a hardware resource is shared among two or more entities, there is a possibility of side and covert channels. The CPU cache hierarchy is a resource commonly used in side channel attacks since in modern CPU designs the L1/L2 caches are typically shared between sibling hardware threads[19] in an SMT architecture, and the L3 cache is typically shared between multiple physical processors. Additionally, software programs scheduled on the same physical CPU effectively share the same cache hierarchy. As memory accesses that hit the cache execute faster than ones who do not, malicious programs can use the cache to communicate or spy on other programs, generally referred to as covert and side channel attacks, respectively.

A variety of research [20] exists around the use of cache-based side channels. The general theme in a side channel attack is that during the execution of the victim program, it perturbs the cache in a way that is noticeable by the attacker. If the way the victim perturbs the cache is directly related to a secret, such as a cryptographic key, the attacker may be able to determine the value of that secret.

Side channel attacks may generally be classified as either architectural or speculative. Architectural side channel attacks involve an attacker determining secrets from the architectural (non-speculative) flow of software execution. Architectural side channel attacks historically have been addressed by software updates, such as by removing secret-dependent memory access patterns[21].

Speculative side channels, such as Spectre[7], Meltdown[22], Foreshadow[23], and MDS[24], [25] rely on the speculative flow of execution in CPU hardware. These attacks leverage features like branch prediction and out-of-order execution and represent a newer class of side channels. In response to these, a variety of mitigations have been deployed including software and hardware updates[26], [27].

In both architectural and speculative cache-based side channels, the attacker determines the victim's secret through the timing of memory accesses. Whether a particular address is or is not cached tells the attacker something about the victim. Because modern CPUs operate at very high frequencies, a very high precision timer is necessary for determining whether a particular address is cached or not. A common technique is to use the x86 Time Stamp Counter (TSC) which counts CPU cycles. By using instructions like `RDTSC`[28] around a memory access, an attacker can get a very accurate timing of how long that access took to execute.

As discussed earlier, instructions like `RDTSC` may not always be available to an attacker, or their use may need to be minimized to avoid detection. In this paper, we demonstrate how CPU speculation can be used to achieve these goals with new signal modification and recovery primitives.

B. CPU Speculation

Speculation is a common technique used in high performance CPU design and involves the CPU starting to evaluate instructions before it is determined whether those instructions are architecturally part of the program flow. A variety of speculation techniques are used in modern microprocessors, including branch prediction[29]. In branch prediction, the CPU hardware guesses how a branch will be evaluated and begins speculatively executing the instructions along the chosen path. If the hardware later determines the guess was incorrect, the architectural effects of the speculative execution are reversed and in-progress instructions are flushed, allowing the correct path to begin execution. However micro-architectural effects of speculation can sometimes remain, including cachelines brought in as a result of speculation.

As modern processors employ out-of-order execution techniques, instructions including branches are evaluated independent of program order. As long as the data dependencies may be satisfied, the branch will be evaluated and potentially redirect the execution of the pipeline. If the data dependencies cannot be immediately satisfied, such as a branch dependent on a load from DRAM, the execution of the branch waits until the required data is available. This behavior is key to the work presented here, as the length of time it takes to evaluate a branch is directly dependent on the time to load its dependent data.

C. Related Work

While much existing research[20] exists on CPU side channels and their applications, there appears to be very little peer-reviewed work in the general area of signal modification or amplification. That being said, in response to the Spectre browser mitigations[13], [14], [15] deployed in early 2018, there have been various workarounds proposed for the lack of high precision timers. These typically include constructing artificial high-precision timers using available resources[30], or calling the victim code multiple times to cause several cache lines to be affected and hitting specific timing windows[31]. Another technique involves using a self-reinforcing loop [32] and CPU speculation to amplify the original signal, while also repeatedly executing the Spectre gadget.

Unlike these techniques, the work presented here does not rely on artificial high-performance timers, nor invoking the victim code multiple times. While invoking the sensitive code repeatedly may be practical for Spectre-based attacks (such as an out-of-bounds array load), in other attacks the attacker may need to hit a very small timing window to observe the victim's behavior. For instance, the MDS attack may need to execute around the exact time when the secret is being used. In these cases, it is likely not practical to attempt to hit this timing window multiple times and instead the attacker must be able to determine as much information as possible from a single invocation. This paper demonstrates methods for amplifying signals to the point that high precision timers are not needed, as well as methods for modifying signals prior to measurement to reduce the number of timing checks needed. These techniques can apply to most types of cache-based side or covert channel attack.

III. Definitions

Throughout this paper, we will refer to cachelines by their virtual address. As this paper focuses on manipulating the state of cachelines, we define $\phi(A)$ to return TRUE (1) if the cacheline at address A is present in any level of the CPU cache hierarchy and FALSE (0) otherwise. More specifically, given a microarchitecture specific threshold value T, $\phi(A) = 1$ if the time required to access A is less than T, and $\phi(A) = 0$ otherwise.

For purposes of signal recovery, the value of the data in the cache typically does not matter and is controlled by the attacker. For instance, in a FLUSH+RELOAD attack, the attacker is interested to know if a certain cache line is present or not, but the value of the data is not relevant. Therefore, in the examples presented in this paper we universally assume that all cachelines signals used contain the value 0. That said all code examples shown here could be modified as needed if this assumption was not possible.

All assembly code in this paper is written in x86 GNU Assembler syntax[33]. In GNU Assembler syntax, the destination is always on the right (e.g., *add src, dest*).

IV. Basic Primitive

A. Theory

In order to manipulate cache signals without reading them, we take advantage of CPU speculation in a controlled manner. The key insight is that we can make the length of speculation completely dependent on the presence (or lack thereof) of one or more cachelines. To demonstrate how this works, let us first examine how to build a simpler inverter.

Assume that initially cacheline B is not present in the cache and the line pointed to by A may or may not initially be present. As noted above, memory at A is assumed to contain the value 0. The most basic form of the signal manipulation primitive is the inverter shown below:

```
if (likely(*A != 0)) {
    Access cacheline B
}
```

```
1: call 3f
2: #Speculative instructions go here
   lfence
3: mov $4f, (%rsp)
   ret
4: nop
```

Fig. 1: Forced Speculation Gadget

The idea is that the CPU is first tricked into mispredicting the branch, resulting in cacheline B being accessed during speculation and potentially brought into the cache. Eventually the CPU will discover that it mispredicted the branch and stop the incorrect speculation. How many cycles it takes for the CPU to discover the misprediction is dependent on whether cacheline A was initially present or not. If cacheline A was present, the branch evaluates quickly and it is unlikely cacheline B will be fetched. But if cacheline A was not present, the branch evaluates slowly resulting in the access to cacheline B.

B. Branch Misprediction

To realize this primitive in practice, there are a few challenges to resolve. First is how to ensure the CPU mispredicts the branch to achieve the desired speculation. Although this could be accomplished through sufficient priming of the conditional branch predictor, in this paper we take advantage of a common and more reliable behavior of most CPUs whereby *ret* instructions are typically predicted to return to the instruction following the previous *call*. This is the same insight that forms the basis for the *retpoline* [34] Spectre mitigation. The code sequence used is shown in Figure 1.

In the architectural flow of this code, the return address on top of the stack is overwritten to point to label 4 so when the *ret* executes, the processor will jump to the *nop* instruction. However, the branch prediction logic will not be aware of this and predict that the *ret* will return back to label 2. The instructions at label 2 will therefore be executed speculatively, and this occurs every time this code is executed without requiring any kind of priming.

C. Inverter

Using the terminology introduced earlier, the function of the signal inverter is that given an input cacheline A and output cacheline B, set $\phi(B) = \neg\phi(A)$. In the code shown in Figure 2, the address of A is assumed to be in RSI while the address of B is in RDI. Note that cacheline B is assumed to initially not be present ($\phi(B) = 0$).

There are two important elements of the Inverter code. The first is that a data dependency is created between cacheline A (RSI) and the top of the stack. The addition of the value of cacheline A (assumed to be 0) does not change the value of the new return address being written, but the CPU does not know

```
1:  call 3f
2:  xor %rax, %rax
    .rept 5 # Delay ops
    mov (%rsp, %rax), %rax
    and $0, %rax
    .endr
    mov (%rdi, %rax), %r11
    lfence
3:  mov $4f, (%rsp)
    mov (%rsi), %r11
    add %r11, (%rsp)
    ret
4:  nop
```

Fig. 2: Inverter

this. It is not able to evaluate the ret until this dependency is satisfied.

The second aspect is the delay loop shown at label 2. This is needed because even if the processor is quickly able to evaluate the ret, some ops may have executed along the incorrect speculative path. This delay loop performs dependent memory operations in order to give the CPU time to redirect itself if the input cacheline happened to be present.

The simplified pipeline diagrams in Figure 3 and 4 demonstrate this behavior. In Figure 3, the execution of the load from RSI happens quickly allowing the ret to flush the pipeline before the load of RDI begins to execute. However in Figure 4, the load from RSI happens slowly, and the RDI load is able to execute.

The result is an inversion of $\phi(A)$. If $\phi(A) = 0$ then the ret is slow and cacheline B is accessed (so $\phi(B) = 1$). But if $\phi(A) = 1$ then the ret is fast and cacheline B is not accessed.

V. Logic Gadgets

This section demonstrates several examples for manipulating the state of cachelines to form standard boolean logic gates, building upon the basic speculation primitive.

A. Replicator

The replicator is similar to the inverter but aims to bring in more than one cacheline based on the input. For instance, with three output lines it sets $\phi(B_1) = \phi(B_2) = \phi(B_3) = \neg\phi(A)$. When accessing the output lines, these instructions are each made to be dependent on the delay loop, but independent of each other. This allows their lines to be fetched in parallel. The code in Figure 5 implements the replicator with an output of three lines. In this example, the target cachelines are pointed to by R8, R9, and R10.

B. NAND Gadget

In the above examples, only a single cacheline served as the input. Next we modify the primitive to look at multiple input cachelines, enabling boolean logic. For instance, the code in Figure 6 sets $\phi(C) = \neg(\phi(A) \wedge \phi(B))$. Cachelines A, B, and C are assumed to be in registers RSI, RDX, and RDI respectively.

In this gadget, the CPU is only able to send the redirect and flush the pipeline if both input cachelines are present. If either one of them is not present, speculation should continue far enough to load the output cacheline.

C. NOR Gadget

As another example, a NOR gate which sets $\phi(C) = \neg(\phi(A) \vee \phi(B))$ can be built using a primitive where either cacheline being present causes the CPU to quickly end the incorrect speculation. In the NOR gate, there are 2 instances of forced incorrect speculation. The first one ($call\ 4f$) will end quickly if $\phi(A) = 1$ while the second ($call\ 5f$) will end quickly if $\phi(B) = 1$. The output cacheline C will only be accessed therefore if both $\phi(A) = \phi(B) = 0$. This gadget is shown in Figure 7. Cachelines A, B, and C are assumed to be in registers RSI, RDX, and RDI respectively.

D. Arbitrary Logic

As both NAND and NOR are universal logic functions, it is possible to implement arbitrary logic functions on the state of cachelines using either of these gadgets. The gadgets shown here may also be trivially extended to provide additional inputs (e.g., a 4-way NAND) or additional outputs. There are limits on how many inputs and outputs may be used in a single gadget in a given microarchitecture and this is discussed more later in section VI.

Note that it is possible to directly build a 2-input XOR gadget using the primitive with three branch mispredictions. It functions by accessing the output cacheline if and only if exactly one of the input cachelines can be accessed quickly. As will be discussed in the evaluation section however, this gadget did not perform as reliably as the other gadgets presented here.

VI. Signal Amplification

Signal amplification refers to the problem of determining the state of a cacheline without access to a high precision timer. In this case, we assume there is an initial cacheline A in an unknown state. Our goal is to manipulate the cache such that when we eventually perform a timing measurement, the timing difference between the case where $\phi(A)$ was 0 vs 1 is as high as possible. We refer to this timing difference as the signal strength.

A. Amplifier Theory

The basic idea of amplification is to make the state of multiple cachelines dependent on the single input. We can then time the access to those multiple cachelines and observe a larger timing difference than the access to the single cacheline.

Let us assume we have an initial cacheline A and two additional cachelines B and C to work with. Using the replicator shown in Section V-A, we can set the state of cachelines B and C to be the inverse of A. To create a noticeable timing delta though, we must be careful in how cachelines are accessed.

In the replicator gadget, cachelines B and C should be accessed independently (meaning the access to cacheline C is

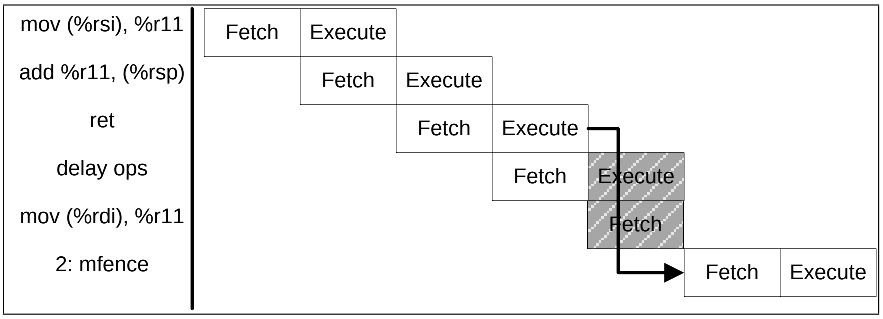

Fig. 3: Pipeline (Cacheline was present)

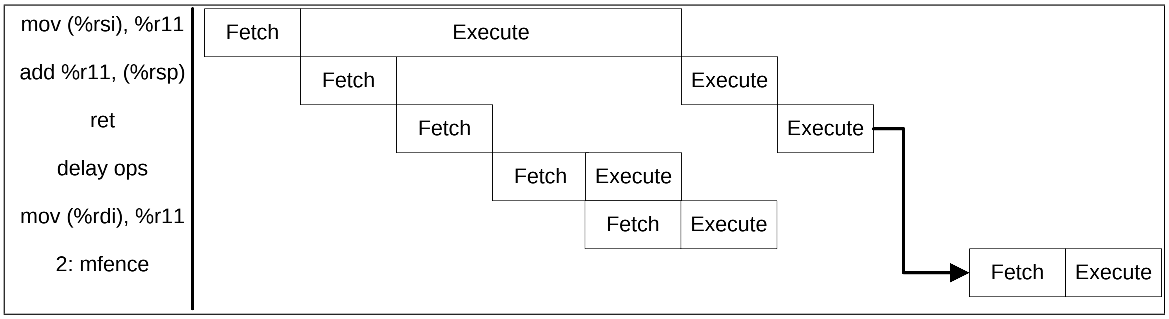

Fig. 4: Pipeline (Cacheline was not present)

```
1:  call 3f
2: <delay ops>
    mov (%r8, %rax), %r11
    mov (%r9, %rax), %r11
    mov (%r10, %rax), %r11
    lfence
3:  mov $4f, (%rsp)
    mov (%rsi), %r11
    add %r11, (%rsp)
    ret
4:  nop
```

Fig. 5: Replicator

```
1:  call 3f
2: <delay ops>
    mov (%rdi, %rax), %r11
    lfence
3:  mov $4f, (%rsp)
    mov (%rsi), %r11
    add (%rdx), %r11
    add %r11, (%rsp)
    ret
4:  nop
```

Fig. 6: 2-input NAND

```
1:  call 4f
2:  call 5f
3: <delay ops>
    mov (%rdi, %rax), %r11
    lfence
4:  mov $6f, (%rsp)
    mov (%rsi), %r11
    add %r11, (%rsp)
    ret
5:  mov $6f, (%rsp)
    mov (%rdx), %r11
    add %r11, (%rsp)
    ret
6:  nop
```

Fig. 7: 2-input NOR

not dependent on the result of the access to cacheline B). This ensures the processor is able to perform both cache accesses in parallel. However when the timing measurement is performed, cachelines B and C should be accessed in a dependent manner. That is, the access to cacheline C *should* be dependent on the access to cacheline B. This ensures that the processor is not able to perform these accesses in parallel, thus exacerbating the timing difference between cache hits and cache misses.

B. Single-Stage Amplifier

The single stage amplifier using a single round of speculation to bring in as many cachelines as possible based on

```
.macro amp deplen accesslen
1: call 3f
2: xor %rax, %rax
   .rept \deplen
   add (%rsp, %rax), %rax
   and $0, %rax
   .endr
   .rept \accesslen
   mov (%rdi, %rax), %r11
   add $STRIDE, %rdi
   .endr
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add %r11, (%rsp)
   ret
4: nop
.endm
```

Fig. 8: Single Stage Amplifier

the input. The amplification primitive is parameterized by two values, $deplen$ and $accesslen$. $deplen$ is defined as the length of the delay loop, while $accesslen$ is defined as the number of output cachelines potentially accessed. In x86 code, this primitive is implemented as shown in Figure 8.

The optimal single stage amplifier will use the smallest practical $deplen$ value with the highest practical $accesslen$ value. The $deplen$ should be as small as possible while still ensuring that if the input cacheline (in RSI) is present, none of the cachelines starting at RDI are accessed. The $accesslen$ value should be as large as a particular microarchitecture can support. As discussed in Section VIII, an $accesslen$ over 20 can be possible.

In the example in Figure 8, the output cachelines are separated by STRIDE bytes. The optimal value of this varies by microarchitecture and is described more in Section VIII.

C. Multi-Stage Amplifier

While a single invocation of the single stage amplifier can affect only a certain number of output cachelines, these amplifiers could be chained together to affect more cachelines to create a multi-stage amplifier. This would theoretically result in the ability to create arbitrarily large signal strengths.

Unfortunately this technique does not work well in real systems because cache sizes are limited. And due to other system activity, it is likely impossible to manipulate the entire state of the cache on a system without interference. In practice, multi-stage amplifiers typically only work with a small number of stages, are subject to more unwanted noise, and generally do not yield signal strengths large enough to measure with extremely coarse timers.

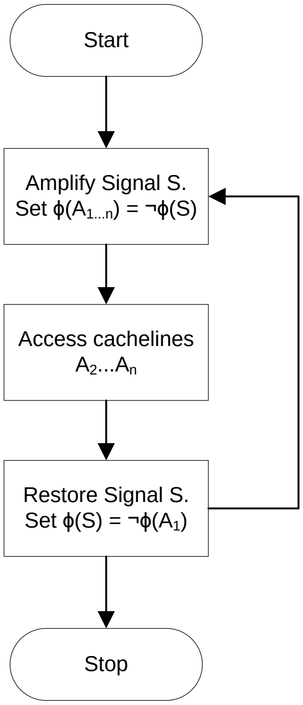

Fig. 9: Self-reinforcing Amplifier

D. Self-reinforcing Amplifier

A more scalable technique to signal amplification is through a self-reinforcing amplifier. The idea of the self-reinforcing amplifier is to amplify the signal using the single stage amplifier, access all but one of the output lines, and finally use the remaining line to restore the original signal as shown in Figure 9. As discussed earlier, during the access stage a dependency on the output cachelines is used to force the timing of the entire loop to become dependent on the original signal. This loop can be repeated many times to create a large signal strength.

VII. Applications

Modifying cache signals prior to reading them can be useful for reducing the number of timer calls needed. This is especially valuable is a high precision timer is either not available or if access to it is being monitored.

A. FLUSH+RELOAD Binary Search

One use case for modifying signals prior to recovery is an efficient FLUSH+RELOAD binary search. The premise is that the attacker is doing a FLUSH+RELOAD attack and has a set of N cachelines, exactly one of which will be brought in by the victim depending on the result of the attack. Each of the cachelines represents a possible output value and each value is equally likely to occur. A naive approach would be to time the access to each of the lines, looking for the one that is fast. However we will show that the attacker can in fact accomplish this with only $log_2(N)$ timed accesses.

An example with N=16 is shown in Figure 10. This algorithm will make use of three types of gadgets. Specifically we

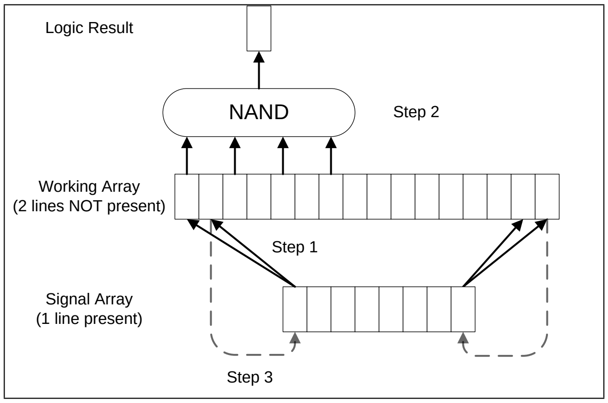

Fig. 10: Binary Search

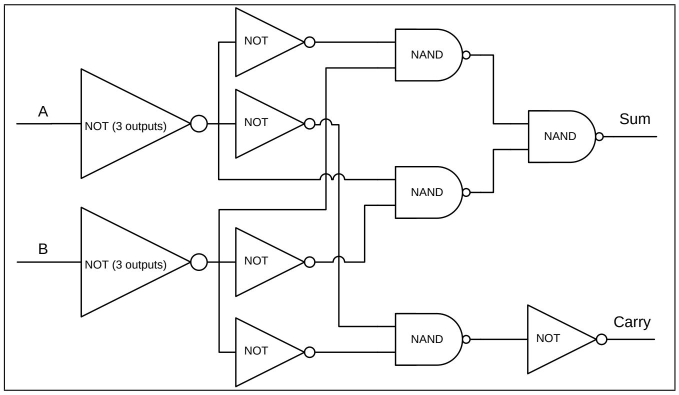

Fig. 11: Half Adder for Cacheline Signals

will use an inverter, a 2x replicator, and a variety of NAND gadgets (8, 4, and 2-input versions). Additionally we will use several memory arrays. The first array, $S_0 ... S_{15}$ is the ”signal” and we assume that exactly one of these cachelines is initially present. The second array W is defined as twice the size of S, from $W_0 ... W_{31}$. The last array is just a single cacheline in size, called R.

On each loop iteration, first the D and R arrays are flushed from the cache. Then the array S is duplicated such that $\phi(W_{2*i})) = \phi(W_{2*i+1})) = \neg\phi(S_i)$, shown as Step 1. Next a test is run to determine which half of the array must be searched. To test indexes x through y we compute a NAND such that $\phi(R) = \neg(\phi(W_{2*x}) \wedge \phi(W_{2*(x+1)} ... \wedge \phi(W_{2*y}))$, shown as Step 2. Finally, we copy the unused lines in W back to S using the inverter such that $\phi(S_i) = \neg\phi(W_{2*i+1})$ as shown in Step 3. This step is necessary because each primitive used is destructive to the input. The replicator helps work around this by keeping an (inverted) copy of the original signal array in odd entries of W.

After each loop, we test if line R is present in the cache or not. If R was present it means that the signal cacheline was in the range $x...y$ and we narrow our search range to those indexes. If R was not present, the signal was in the other half.

As with a standard binary search, this algorithm is able to determine which of N cachelines was present using only $log_2(N)$ timing checks.

B. Cacheline Counter

Another use case that can be optimized is when an attacker has an array of cachelines which may or may not be present, and they want to know how many are present. This might occur if the attacker is doing a FLUSH+RELOAD attack and the victim will bring in different numbers of lines based on what code it executed.

Using the logic gates described earlier, it is possible to build an adder where the state of a set of cachelines represent the counter value. For example, the attacker may have an array of 64 cachelines, in which case a 6-bit adder is needed. The state of this adder is represented by six cachelines, each corresponding to a bit. The attacker then takes each of the set of input cachelines and performs a binary addition between that input cacheline and the 6-bit adder.

The binary addition is performed using the logic gadgets introduced earlier. In particular, an inverter, 3x replicator, and 2-input NAND are used to build a half-adder as shown in in Figure 11. Note that this logic is slightly different from a standard half-adder because of the limitations of using cache line state. Each use of a particular cache line is destructive so copies of each input must be made at the beginning. Additionally, as shown earlier it is easiest to build gates like NAND instead of XOR, so the half-adder is written exclusively using NAND and inverter gadgets.

After the state of all the input cachelines have been added together, the attacker can read the value of the counter by timing accesses to it. In the example above, the attacker therefore only needs to perform six timing measurements to determine how many of the 64 cachelines were present.

VIII. Evaluation

An AMD Threadripper™ 3970X running Ubuntu 20.04 and Linux® kernel 5.10.0-rc7 was used for testing. This system has 32 CPU cores (64 threads with SMT) each with 32kB of L1 cache (8-way set associative), 512kB of L2 cache (8-way set associative), and 128MB of L3 (16-way set associative). The cacheline size at all levels of the hierarchy is 64 bytes. The L1 and L2 caches are private to each core on this CPU while the L3 cached is shared amongst groups of eight cores. During testing, all code was pinned to a single CPU core. Note that while an AMD processor was used for evaluation, the techniques described in this paper are generic and are likely to behave similarly on processors from other vendors as well.

In all tests, different cachelines were configured to be at least 4160 (4096+64) bytes apart. Keeping cachelines at least 4096 bytes apart was necessary to avoid unwanted interaction from hardware prefetchers. Offsetting cachelines by an additional 64 bytes ensures that different cachelines would reside at different L1 indexes to avoid contention as the L1 index uses bits 11:6 of the virtual address.

Gate	Inputs	Correct Results	Accuracy
NOT	1	799998	100%
NOR	2	799998	100%
NAND	2	799996	100%
NAND	4	799996	100%
NAND	8	799999	100%
NAND	16	799995	100%
NAND	32	799995	100%
NAND	64	799941	99.99%
NAND	128	799649	99.96%
XOR	2	772869	96.61%
Half Adder	2	799994	100%

TABLE I: Logic Gate Results (800k runs)

A. *Logic Gates*

A special testbench was created to test the reliability of the basic logic gate gadgets. This testbench runs each each set of possible inputs to each logic gate for a total of 800k runs. After each run, the output cachelines were checked to determine if the logic gate performed correctly. The results are shown in Table I.

The accuracy of the logic gates was generally very high. The worst performing gate was the native XOR gate. The native XOR gate struggled to consistently produce the correct result in the $\{0, 0\}$ input case. We believe this is because if one cacheline is returned significantly faster than the other for some reason, it effectively aliases to either the $\{0, 1\}$ or $\{1, 0\}$ case. For this reason, the half adder logic used NOT and NAND gates instead as described in section VII-B.

B. *Amplifiers*

To determine the optimal parameters for the amplifier, testing was performed to determine the how quickly the CPU could resolve speculation and how many different cachelines could be accessed during speculation. On the AMD Threadripper CPU used, it was determined that a $deplen$ (delay) value of 5 and an $accesslen$ value of 23 worked well. In other words, in the single stage amplifier, 23 output cachelines were set to the inverted state of the input cacheline.

Each of the 23 output cachelines were exactly `STRIDE` bytes apart. As mentioned earlier, this value was set to 4160 bytes to avoid L1 index contention as well as to avoid the hardware prefetcher accessing cachelines on its own.

The single stage amplifier was subsequently used to create a self-reinforcing amplifier. The self-reinforcing amplifier was run for various numbers of iterations to determine the best possible signal strength as shown in Figure 12.

While the self-reinforcing amplifier could theoretically be run as many times as desired, the data indicates that the signal strength steadily rises until around 700k iterations after which additional iterations do not provide consistent benefits. We believe this is because the gadgets used are subject to some risk of errors, such as due to system interrupts. If the original signal gets corrupted, then the self-reinforcing amplifier will cease to function properly. As the iteration count increases, the chance of signal corruption similarly grows.

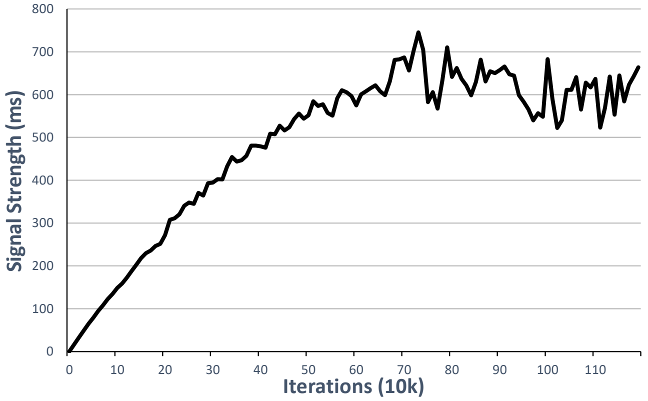

Fig. 12: Amplifier Results (2000 runs each)

Despite the risk of signal corruption, the results demonstrate that the self-reinforcing amplifier performs extremely well and is capable of amplifying signals above $600ms$. As a point of reference, on the tested platform a non-amplified signal had a strength of approximately $76ns$ (as measured by the TSC). This means that the amplification logic achieved a $8 * 10^6$ amplification on average.

While the average results of signal amplification are significant, the consistency of the algorithm was also evaluated. Although average signal strength rises with higher iteration counts, there is also a wider range of results and higher risk of incorrect amplification. Comparing the graphs of Figures 13 and 14 shows how despite 700k iterations yielding larger signal strengths, the results from 100k iterations were less subject to errors as indicated by fewer negative signal strengths.

The consistency of various iteration counts are shown in Table II. To determine the chance of correct signal recovery, the number of runs that demonstrated a signal above the given timer threshold were subtracted from the number of runs that demonstrated an incorrect signal below the same threshold. For instance, in the 100k configuration, out of the 1000 runs tested, a signal strength above $10ms$ was found in 964 runs. But an incorrect signal below -$10ms$ was found in 11 runs. In the remaining 25 runs the signal was less than $10ms$ and thus not able to be measured with a $10ms$ timer. This result is reported in Table II as a 95% chance of correct signal recovery.

The chance of an indeterminate signal was defined as a result where the absolute signal strength was less than the granularity of the timer. For example, in the 300k configuration with a $100ms$ timer, there is an 18% chance that the difference between the source cacheline being present or not present was measured as being less than $100ms$.

The results show that the optimal choice of amplifier configuration is highly dependent on the available timer. With a $10ms$ timer the 100k configuration performs best among the ones tested as there is a 95% chance of correct signal recovery. On the other hand if only a $500ms$ timer is available then the 700k amplifier is the best choice. Although it only recovers the signal correctly 50% of the time, the result is indeterminate another 40% of the time. The chance of an incorrect signal recovery is thus only 10%. If it is possible to conduct the attack multiple times, the chance of correct signal recovery

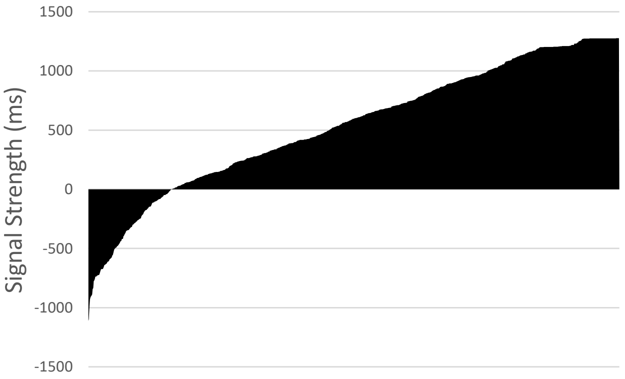

Fig. 13: 700k Iteration Amplifier Results (1000 runs)

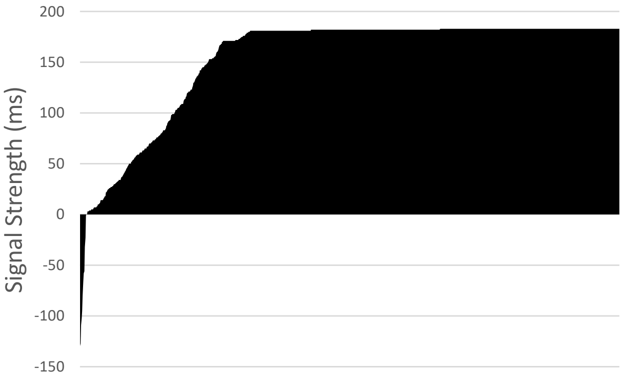

Fig. 14: 100k Iteration Amplifier Results (1000 runs)

Amplifier Configuration	**100k**	**300k**	**500k**	**700k**
Average Signal Strength (ms)				
1st Quartile	155	112	175	155
Median	182	293	484	596
3rd Quartile	183	512	827	990
Chance of Correct Signal Recovery				
With 500ms timer	0%	26%	45%	50%
With 100ms timer	82%	71%	72%	66%
With 10ms timer	95%	80%	75%	68%
Chance of Indeterminate Signal				
With 500ms timer	100%	74%	50%	40%
With 100ms timer	17%	18%	11%	8%
With 10ms timer	3%	2%	1%	1%

TABLE II: Amplifier Consistency (1000 runs)

Size	Correct Results	Accuracy
4	100000	100%
8	99999	100%
16	99995	100%
32	99999	100%
64	99991	99.99%
128	92374	92.37%
256	66398	66.40%

TABLE III: Binary Search Results (100k runs)

will increase.

C. Binary Search

The binary search algorithm described in section VII-A was implemented in various sizes using a similar testbench. The testbench picks a random line in the set and runs the binary search algorithm to determine if it could accurately pick out the single present cacheline. The results are shown in Table III.

The binary search algorithm was very accurate for all sizes through 128 but did drop off in the 256 configuration. With larger sizes the algorithm does run longer and this increases the risk of errors.

The performance of the binary search algorithm is slower than the simple approach of testing each cacheline sequentially by about 15-20x. However, if the cost of measuring a cacheline is expensive (such as if amplification is required before the cacheline can be measured) then the binary search algorithm is a useful optimization as many fewer measurements are required.

D. Cacheline Counter

The cacheline counting algorithm described in section VII-B was also implemented in various sizes using a similar testbench. The testbench randomly initialized the set of cachelines so some were and some were not present and then ran the cacheline counting algorithm to determine if it could correctly count the number of present cachelines. For instance, for size 16 then a set of 16 cachelines were randomly initialized in various states and five cachelines was used to capture the 5-bit count result. The results are shown in Table IV.

The cacheline counter algorithm was highly accurate for the sizes tested, although as expected accuracy did decline slightly at larger sizes.

IX. Mitigations

As this paper does not demonstrate a new attack, there is likely no need for mitigations specific to the gadgets discussed here, and such mitigations are unlikely to easily exist. Many of the existing mitigations for side channel attacks like Spectre rely on preventing the undesired behavior in the first place[35], [36], [37], including by preventing the attacker from mounting such an attack or isolating memory so it cannot be read even during speculation[38]. These mitigations are effective and should remain as the primary defense.

The techniques demonstrated here do heavily rely on branch misprediction and as such, it may be possible to detect their use through performance monitor counters or other tools that can detect large amounts of mispredicted branches. The challenge with this approach is that mispredicted branches are not uncommon and being able to reliably differentiate between attack code and benign code may be quite difficult.

As this paper demonstrates, reducing the precision of timers available to an attacker does not render cache-based side channel attacks impossible, but it does require extra work to employ techniques like signal amplification. And if an attacker is required to use signal amplification, the performance of their attack may drop dramatically as each signal may take milliseconds to read instead of nanoseconds. Therefore, re-

Size	Correct Results	Accuracy
4	99991	99.99%
8	99981	99.98%
16	99937	99.94%
32	99842	99.84%
64	99697	99.70%
128	99275	99.28%
256	97982	97.98%

TABLE IV: Cacheline Counter Results (100k runs)

ducing the precision of timers may still be a useful defense-in-depth strategy, but it does not eliminate the threat of these side channels attacks.

X. Future Work

Optimizing cache signal recovery is rather unexplored area, and future research is needed to determine the practical limits of signal modification and amplification in other contexts, including browsers, as well as evaluation on non-x86 microarchitectures.

Beyond side channel attacks, computation on the presence of cache lines could be an interesting new paradigm for confidential computing. As secret data can be stored in the state of cachelines instead of in registers or memory, this information would be invisible to a higher privilege entity like an operation system. Furthermore, attempts to conduct side channel attacks *against* this type of software may prove futile as any modification of the cache state may corrupt the data being used by the program. As such, a program would be aware if someone attempted to read its state, which is a potentially unique security property.

Finally, error detection and correction with cache side channel signals may be a useful area of exploration. Although each operation on cache state carries with it some risk of inaccuracy, this may be able to be mitigated using error correction techniques.

XI. Conclusion

This paper demonstrates a number of new speculation-based gadgets for optimizing and amplifying the signals from cache-based covert and side channel attacks. Being able to modify signals without measuring them is a new enabling technology for advanced side channel research, especially in cases where signals are either difficult to measure due to a lack of high precision timer sources, or if access to the timer may be monitored by an adversary. Algorithms such as the binary search and cacheline counter can be employed to reduce the number of timing measurements needed by the attacker, thus improving overall efficiency or avoiding detection.

For real-world scenarios involving attacks which cannot be easily repeated (such as ones requiring specific timing windows), the techniques shown in this paper are likely to be especially effective as signal modification and amplification can be achieved without having to invoke the victim multiple times. A single signal can be amplified to the point of being measured reliably with timers as coarse as $100ms$ or more, which are typically easily available.

Understanding these techniques is important for both attackers and defenders, and will hopefully lead to improved mitigations as well as additional research in the area of side channel signal recovery.

References

[1] K. Schramm, G. Le, P. Felke, and C. Paar, “A collision-attack on aes combining sidechannel- and differential-attack,” in *CHES’04. Volume 3156 of LNCS*. Springer, 2004, pp. 163–175.

[2] O. Acıiçmez, W. Schindler, and Ç. K. Koç, “Cache based remote timing attack on the aes,” in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 271–286.

[3] A. Bogdanov, “Improved side-channel collision attacks on aes,” in *International Workshop on Selected Areas in Cryptography*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 84–95.

[4] ——, “Multiple-differential side-channel collision attacks on aes,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 30–44.

[5] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley, “Cache-timing attacks on rsa key generation,” Cryptology ePrint Archive, Report 2018/367, 2018, https://eprint.iacr.org/2018/367.

[6] J. Horn, “Reading privileged memory with a side-channel,” Jan 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[8] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[9] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, “A high-resolution side-channel attack on last-level cache,” in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC ’16. New York, NY, USA: ACM, 2016, pp. 72:1–72:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2897962

[10] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, “A new prime and probe cache side-channel attack for cloud computing,” in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, Oct 2015, pp. 1718–1724.

[11] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[12] M. Lipp, V. Hadzic, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, “Take a way: Exploring the security implications of amd’s cache way predictors,” in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[13] J. Hazen, “Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer,” Jan 2018. [Online]. Available: https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/

[14] Mozilla, “Mozilla foundation security advisory 2018-01,” Jan 2018. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/

[15] F. Pizlo, “What spectre and meltdown mean for webkit,” Jan 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[16] A. Fog, “Cache side channel attacks: Cpu design as a security problem,” in *Cache side channel attacks: CPU Design as a security problem*, Hack In The Box. Amsterdam: HITBSecConf, 2016, https://conference.hitb.org/hitbsecconf2016ams/materials/D2T1%20-%20Anders%20Fogh%20-%20Cache%20Side%20Channel%20Attacks.pdf.

[17] R. Martin, J. Demme, and S. Sethumadhavan, “Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 118–129. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337173

[18] D. Katzman, W. Kosasih, C. Chuengsatiansup, E. Ronen, and Y. Yarom, "The gates of time: Improving cache attacks with transient execution," in *USENIX Security*, 2023.

[19] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty *et al.*, "Hyper-threading technology in the netburst® microarchitecture," *14th Hot Chips*, 2002.

[20] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, 03 2018.

[21] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[23] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [39].

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

[25] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 769–784. [Online]. Available: https://doi.org/10.1145/3319535.3363219

[26] Intel, "Intel analysis of speculative execution side channels," Jan 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[27] AMD, "Software techniques for managing speculation on amd processors," Jul 2018. [Online]. Available: https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf

[28] ——, "Amd64 architecture programmer's manual volume 3: General-purpose and system instructions," May 2018. [Online]. Available: https://www.amd.com/system/files/TechDocs/24594.pdf

[29] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 135–148.

[30] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Financial Cryptography and Data Security*, A. Kiayias, Ed. Cham: Springer International Publishing, 2017, pp. 247–267.

[31] N. H. . J. Afek, "Overcoming (some) spectre browser mitigations," June 2018. [Online]. Available: https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/

[32] Anonymous, "Spectre cascade – there may be no safe timer mitigation," Feb 2018. [Online]. Available: https://weblll.org/index.php/spectre-cascade-there-may-be-no-safe-timer-mitigation/

[33] Wikibooks, "X86 assembly/gas syntax," May 2018. [Online]. Available: https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

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

[35] M. Miller, "Security research & defense information from microsoft about vulnerabilities, mitigations and workarounds, active attacks, security research, tools and guidance mitigating speculative execution side channel hardware vulnerabilities," Mar 2018. [Online]. Available: https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

[36] B. Stuart, "Current state of mitigations for spectre within operating systems," *Advanced Microkernel Operating Systems*, p. 47, 2018.

[37] C. Carruth, "Speculative load hardening," Dec 2018. [Online]. Available: http://llvm.org/docs/SpeculativeLoadHardening.html

[38] C. Reis, "Mitigating spectre with site isolation in chrome," July 2018. [Online]. Available: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html

[39] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [23].