Spectre Without Shared Memory

Ben Amos
Ben-Gurion University of the Negev
Be'er Sheva', Israel
amosbe@post.bgu.ac.il

Niv Gilboa
Ben-Gurion University of the Negev
Be'er Sheva', Israel
gilboan@bgu.ac.il

Arbel Levy
Ben-Gurion University of the Negev
Be'er Sheva', Israel
levyarb@post.bgu.ac.il

ABSTRACT

The Spectre attack by Kocher et al. [10] reads arbitrary data from colocated processes by exploiting two common features of modern processors: speculative execution and shared caches. While theoretically the attack works in many different settings, the current variations all require that the attacker share with the target a memory region that includes vulnerable code which accepts input from the attacker.

Motivated by the common practice in cloud computing of *not* allowing shared memory between different users, we construct the first Spectre type attack in which the target and the attacker *do not* share any memory pages. The target is a server and the attacker is colocated with the target, shares a Last-Level Cache with it and provides input to the target as a typical client over TCP.

We develop new techniques for the attack including accurate location of the target's code and data in the shared cache, noise suppression enabling reliable retrieval of the target's data and optimizations speeding up the retrieval process. An indispensable tool in the retrieval process is a careful comparison of cache activity between two scenarios: the attacker sending as input an address of interest x and the attacker sending a different address x'. The comparison enables extraction of a single memory byte from the target.

We report on a Proof-of-Concept implementation of our attack and on tests on two Intel multi-core platforms with inclusive Last-Level Caches and speculative execution. The tests ran in two virtualization settings, Virtual Machines and Linux containers and in two profiles of cache activity, relative inactivity and very high activity. The setup phase in which the attacker locates the target's data in the cache requires on the order of several minutes to several tens of minutes. The attack successfully extracts the data with probability per byte between 0.91 to 0.99 and rate ranging from 0.4 to 10 bytes per second.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Malicious design modifications**;

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297470

KEYWORDS

Side-channel attack, Cross-VM side channel, Last-level cache, Speculative execution

ACM Reference Format:
Ben Amos, Niv Gilboa, and Arbel Levy. 2019. Spectre Without Shared Memory. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3297470

1 INTRODUCTION

Designers use many strategies to maximize the performance of processor chips including prediction of future control flows and of data required in future computation. Two important predictive components are fast cache memories and speculative execution. Cache memories are relatively small memory elements which enable faster access time than main memory. In speculative execution, if the processor does not have all the information necessary to continue its execution it guesses the control flow and continues execution instead of stalling. If the prediction is correct then the processor saves time, while if it is incorrect then the processor rolls back to its initial state and continues with the correct execution flow. Speculative execution can also be performed by executing two or more control flows concurrently but retiring only one when the correct flow can be determined.

Predictive components typically change the running time of a program based on its input and state. An attacker who shares such resources with a target may observe them to deduce information on the target. The attacker's general method is to create contention with the target over a shared hardware resource that influences execution time. By measuring the time required for its *own execution* the attacker can infer information on the target's use of the shared component which implies information on the target's input.

Attacks based on timing side-channels in processors' micro-architecture have exploited both cache memory [3, 7, 9, 13, 15, 18, 21, 22, 24] and speculative execution [2]. Side-channel attacks based on micro-architecture differ in the degree of sharing that they require between the attacker and the target. If the attacker and the target execute on the same core then they share all the levels of cache memory and in addition processor components that support speculative execution such as branch predictors. In this case, all these components can potentially be exploited for side-channel attacks. However, if the two processes execute on separate cores then typically the only shared hardware resource is the Last Level Cache (LLC) which is shared among all cores. A different type of resource that is sometimes shared is part of the memory space of each process. This is most often the case when the two processes run the same software library and the underlying operating system saves memory by using deduplication to load the same (typically read only) memory pages once for both processes instead of twice.

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

Figure 1: Code fragment vulnerable to Spectre attack [10].

Cloud services are the most common ecosystem in which Virtual Machines (VM) or Containers, provided by different users share the same underlying hardware. Attacks by Ristenpart et al. locating a target VM in the cloud [16] and extracting information from it spurred interest in a variety side-channel attacks exploiting shared resources to obtain secret data from victim VMs.

Cloud providers enable co-tenancy of VMs from different users for economic reasons. Each VM is assigned a virtual CPU, which is implemented as a series of time slots on a physical CPU. It is considered unlikely that a cloud provider will run VMs from different users on a single core allowing single core attacks. In addition, due to security concerns most cloud providers disable deduplication of memory pages [19, 20] between VMs. Therefore, micro-architectural side-channel attacks in cloud environments need to be cross-core and not dependent on shared memory.

The Spectre attack by Kocher et al. [10] and its close relative, the Meltdown attack by Lipp et al. [12] built on earlier work by Gruss et al. [5] and exploiting both speculative execution and cache memory to extract *arbitrary data* from the victim given very reasonable assumptions. The two attacks differ in their target with Meltdown enabling a user process to extract data from its kernel's memory space and Spectre allowing a user process (or a VM) to extract data from another process or another VM.

In Spectre the target code includes a branch such as the example in Figure 1 where the attacker controls x. The attacker trains the branch predictor to enter the if statement by sending legal values of x and then causes it to speculatively enter the statement for an illegal x which allows the attacker to obtain the value of array1[x] by a cache side-channel attack.

While the Spectre attack can theoretically be used in a cloud setting, the proof-of-concept provided with the attack uses shared memory between the attacker and the target in a fundamental way. Sharing the variable array1_size and the addresses of array1 and array2 enable the attacker to both identify all the necessary locations in the cache without error and use the low noise Flush+Reload attack [24] to extract data from the cache. However, as previously noted, the most important setting for this type of attack is cloud computing in which such memory sharing is highly unlikely.

In this paper we fill the gap and construct a fully operational Spectre attack that does not assume shared memory between the attacker and the target. A naïve approach to the problem of constructing Spectre without shared memory is to replace the Flush+Reload attack with a cache attack that functions without shared memory such as Prime+Probe of the LLC [13] and run the other parts of the attack without modification. However, this approach fails in several ways. Without shared memory the attacker does not know the addresses or corresponding cache sets of array1_size and array2. Without this information the attacker cannot flush array1_size from the cache, which is crucial for triggering speculative execution, and does not know which cache sets to test for activity. Furthermore, the naïve approach is susceptible to significant noise that is exactly correlated with sending an input x because it involves code and data that the attacker and target use for sending and receiving x. It is therefore highly likely that incorrect cache sets exhibit the same behavior as the set holding array2[array1[x] $*$ 256] leading to decoding errors.

Contribution. We make two contributions in this work. The first is the construction of a Spectre type attack retrieving arbitrary data from a target that does not share any memory region with an attacker. The second contribution is a Proof-of-Concept implementation of the attack which we provide together with data on its probability of success rate of data extraction.

The attack is divided into two phases, a setup phase and the data retrieval phase. In both phases the attacker transmits pairs of different inputs x and x' and by carefully comparing the cache activity for each transmission deduces the location of memory elements or their value.

In the setup phase, the attacker locates the cache sets that store array1, array2 and array1_size by using the technique of pair transmission. It uses several pairs to collect some of the cache sets holding addresses in array2 and then infers the rest of the locations of array2. The attacker then finds the cache set holding array1_size by a combination of brute force search on the cache sets, pair transmission and triggering speculative execution in the target.

Following the setup, the attack extracts the value of array1[x] by using pair transmission in which x indicates the address of interest and x' is a legal address within array1. The attack requires a noise suppression method that increases the signal to noise ratio in the measurements of cache activity. The attack uses several optimizations including weighted sampling of cache sets. The attack estimates the probability of a set holding the address of array2[array1[x] $*$ 256] and samples sets that are likelier to hold the address more frequently than others.

We implement a PoC of our attack [1] and evaluate it on two different Intel chips, on both VMs and containers and in two different cache activity profiles: low activity and very high activity. We measure the rate of data extraction from the target and the probability of each retrieved byte being correct. The extraction rate we report ranges between about 0.4 byte per second to ten bytes per second while the accuracy ranges between 0.91 to 0.99. This extraction rate is still significantly lower than Spectre with shared memory due to several factors. First and foremost is our usage of TCP and virtualization which are together the bottleneck in this attack. The second factor is that Flush+Reload is much faster than Prime+Probe.

2 BACKGROUND

2.1 System Model

Cache memory. Cache memory in modern, multi-core computers typically consists of a hierarchy of memory elements in which the higher levels of the hierarchy are smaller, faster and are associated with a single core. The lowest level of the hierarchy, the Last Level Cache (LLC), is the largest and slowest of the elements and is shared among all the cores. Despite being logically shared, beginning with Intel's Sandy Bridge architecture [13] the LLC in Intel designs is physically divided into *slices*, one slice per core. Every core can access all the slices but retrieval time is proportional to the distance between the core and the cache slice.

Memory is arranged in *lines* of B bytes, typically $B = 64$. A read or write instruction causes the hardware to retrieve the corresponding memory line from the nearest cache in which it resides or from main memory if the line is not found in any cache. A line retrieved from main memory is stored in all levels of the cache for quick future access.

Contention in the cache is inevitable due to the small size of all cache levels compared to main memory. A common approach to deal with contention is *set associative* caching. In this architecture a cache is divided into sets of lines, each memory line is mapped to a cache set and each set stores up to w lines of memory concurrently. Attempting to store excess lines in a full set triggers line eviction in which one or more lines are removed from cache according to a *cache replacement policy*. Intel publicly states that the cache replacement policy in its CPU chips is "pseudo Least Recently Used (LRU)" without giving full details on the policy.

In Intel architectures, the lower $\log B$ bits of a memory address identify its offset within a memory line. If a slice of the LLC contains L cache sets then the next $\log L$ bits of the address are the *set index* identifying the set in an LLC slice in which the line is stored. Finally, the rest of the address bits are the *tag*. Intel architectures use an undocumented hash function to map a memory address to a slice of the cache. According to Liu et al. [13] (refining previous observations by Hund et al. [7]) the hash function takes as input the tag bits if the number of cores is a power of two and otherwise takes both the tag and the set index as input.

An important property of the cache hierarchy is *inclusiveness*. In inclusive caches a lower level, larger cache, holds a strict super-set of the items in a higher level cache. A consequence of inclusive caches is that evicting a cache line from the shared LLC initiates the invalidation of the corresponding lines from the other cache levels. In most Intel architectures the LLC is inclusive, with the notable exception of the recent Skylake-server architecture [8].

Speculative execution enables processing instructions even when the execution path depends on data that the processor does not possess. Consider for example a conditional branch that depends on uncached data. A processor can speculatively choose a branch and continue execution while in parallel retrieving the data from memory and saving a *checkpoint* allowing it to roll back to a previous state if the speculation was wrong, with no performance penalty compared to idling until data arrives. If the speculation is correct then the processor uses the cycles needed to retrieve the data to move forward in its execution path. Speculative execution is supported by several hardware components including the Branch Prediction Unit (BPU). The BPU stores information on whether branches were previously taken in a Branch Target Buffer (BTB) and predicts whether a branch will be taken in the future based on these previous execution paths.

2.2 Cache Attacks

Cache side-channel attacks exploiting the measurable timing differences between access to main memory and access to cache memory were used by Tsunoo et al. [18] to retrieve DES keys and by Bernstein [3] and Osvik et al. [15] to extract AES keys. The work of Ristenaprt et al. [16] on locating a target VM and conducting side-channel attacks in the cloud, was followed by cache side-channel attacks against a variety of targets including recovering cryptographic keys, [6, 9, 13, 24], defeating Address Space Layout Randomization (ASLR) [5, 7] and discovering which web pages a user is browsing [14].

Cache side-channel attacks are based on a number of techniques including *Prime+Probe* and *Flush+Reload*. In a *Prime+Probe* attack [15] the attacker *primes* a portion of the cache by filling it with memory lines from the attacker's memory space, and after an appropriate period of idling the attacker *probes* the cache by reading the same memory lines. If during the idle period the victim accesses a memory line (in its own space) that is mapped to the primed cache area, that line will likely evict one of the attacker's lines, allowing the attacker to deduce this event by timing each of its probes. In a *Flush and Reload* attack [24] the attacker and the victim share the same memory page. The attacker *flushes* a shared memory line, e.g. by using the `clflush` instruction, idles and then reloads the same memory line. The attacker can deduce victim access to the line by measuring the required time for reload. Flush+Reload is faster and more accurate than Prime+Probe but is only applicable in a shared-memory setting.

2.3 Spectre

There are currently several known variants of a Spectre including variants 1 and 2 in the Spectre paper [10]. See [4] for a classification and overview of Spectre exploits. In this paper we focus on Spectre variant 1 which is arguably the most intractable of all variants.

Several basic assumptions are required for Spectre variant 1: speculative execution must be allowed in the victim system, the attacker and the target must be co-resident on the same physical system, allowing the attacker to observe the effects of speculative execution and the target must include a vulnerable code snippet, such as the code in Figure 1, with the attacker supplying the input x. In that example array1 is defined as an array of bytes of length array1_size and array2 is an array of bytes in which there are 256[1] bytes for every byte of array1.

The goal of the attacker is to read the value array1[x] for any offset x, regardless of whether that address is within the bounds of array. The attacker trains the branch predictor by sending several legal values of x, which cause the execution path to continue to the assignment to y. The attacker then proceeds to flush the memory line that contains array1_size and then sends the desired offset x. The target's speculative execution causes a misprediction and the execution continues with an illegal assignment to y, which is not discovered until array1_size is retrieved from main memory. The assignment causes the target to read the address array2[array1[x] * 256] and store that value in the LLC. A standard Flush+Reload attack is sufficient to test the collection of all the possible cache sets in which array2[array1[x] * 256] is stored.

Prototype implementations of Spectre variant 1 for several different settings appear in [10] including a C implementation for the setting of an attacker process running natively on the victim host and sharing the vulnerable memory region with the victim process.

[1] While we use the same code fragment as the Spectre paper, our tests verify that the value 256 can be reduced to 64. Any further reduction results in array2[array1[x]] being mapped to the same cache line as array2[array1[x']] for array1[x] $\neq$ array1[x'] and therefore the attacker cannot assign values to array1[x] with certainty.

The interest in the Spectre and Meltdown attacks has already resulted a significant body of work including automatic verification tools that identify the necessary conditions for conducting such attacks [17]. This work achieves Spectre using Prime+Probe, as we do in this current work, but uses *shared* memory to do so, unlike our attack.

3 THE ATTACK

3.1 Key Components and Assumptions

The attacker has two major components. The first is a *send* process that transmits values x to the victim in order to change the state of the cache sets that store array1[x] and array2[array1[x]*256]. The second is a *receive* process that performs a cache side-channel attack to recover the value of array1[x] by determining the cache set storing the line of array2[array1[x]*256]. The two processes can be implemented as Virtual Machines, Containers or standard user-space processes. We assume that each process runs on a dedicated core and executes in parallel. The receive process must be colocated with the victim. Although the send process may be theoretically located elsewhere, we assume that it is also located on the same machine to facilitate synchronization. We further assume that the receive process has access to a fine-grained clock, such as Intel's `rdtscp` instruction. We denote an invocation of this clock by $time()$.

Similarly to [13] we assume that the memory of the receive process is arranged in huge pages, which is the standard practice in cloud computing. In the Proof of Concept attack we implement, the victim and the receive process execute on a computer with a number of cores that is a power of two. That assumption simplifies the setup process of the attack in Section 3.3, but can be dispensed with at the cost of longer setup time.

Similarly to [10] we assume that the victim code contains a code fragment of the type presented in Figure 1, that the victim enables speculative execution and that the attacker controls the x input in that code. Unlike [10] and [17] we do not assume that the attacker's *shares memory* with the victim or has knowledge on the victim's code or data except for the existence of the code fragment.

3.2 The Communication-Channel Approach

The attack can be abstractly modeled as a communication system in which the victim is a sender, transmitting data to the attacker. In more detail, consider a sender and a receiver who have a synchronized clock and share 256 ordered communication sub-channels which at each time unit can be either active or inactive. The sender can activate a sub-channel and the receiver can test a sub-channel for activity. The sender transmits a byte b by activating the corresponding sub-channel and the receiver obtains the byte by testing every sub-channel for activity.

In our proposed attack, the sub-channels are the cache sets that store the memory addresses array2 + 256b for $b = 0, \ldots, 255$. The victim signals on sub-channel b = array1[x] by speculatively accessing array2[b∗256]. The attacker tests a sub-channel b for activity by running Prime+Probe on the cache set that stores array2[$b * 256$].

A basic tool for transmission and reception is the *LocateSets* algorithm described in Figure 2 in pseudo-code. The algorithm makes no assumptions about the victim beyond the assumptions of Section 3.1 and is therefore especially useful for setting up the communication system.

LocateSets$(S, p, w, n, interval, x, x', \alpha)$

(1) For each $s \in S$ do $score(s) \leftarrow 0$
(2) Let $\{S_1, \ldots, S_{|S|/p}\}$ be a partition of S into subsets of size p.
(3) For each $\tilde{S} \in \{S_1, \ldots, S_{|S|/p}\}$ run simultaneously
  (a) Send(w, n, x, x')
  (b) Receive$(w, n, interval, \tilde{S})$
(4) Let $S' \leftarrow \{s \in S \mid score(s) > \alpha\}$.
(5) Return S'

Figure 2: Finding a collection of cache sets S' that are correlated to an input x.

The goal of LocateSets is to discover cache sets s that are *correlated* to an input x, i.e the state of s changes when x is sent, but does not change when $x' \neq x$ is sent. LocateSets accepts as input a collection of cache sets S, inputs x, x' and a threshold $0 \leq \alpha \leq 1$ and assigns a score in the range $[0, 1]$ to every $s \in S$. It returns a sub-collection $S' \subseteq S$ of cache sets with a score greater than α.

LocateSets initiates two processes that run in parallel and are described in Figure 3. The *Send* process takes as input two different indices x, x', a time parameter w in units of clock cycles and a number of repetitions n. The basic procedure of the Send algorithm is to transmit x repeatedly for a period of w clock cycles and then transmit x' for another w clock cycles. This basic procedure is repeated n times allowing the Receive process to obtain accurate measurements.

The *Receive* process takes as input a collection of cache sets $\tilde{S}$, the same w and n parameters as Send and a time parameter $interval$ which is an upper bound on the time between consecutive transmissions of a value x (or x') and the time required for the victim to process the input, reaching the code fragment that affects the cache. Receive estimates the correlation of a set $s \in \tilde{S}$ to x by repeatedly probing the set and comparing the number of misses during the transmission periods, i.e. the miss count during the transmission of x to the miss count during the transmission of x'. The idea is that any cache set that is correlated with x will have more cache misses in the first period than in the second. Note that sets involved in transmission and reception of data regardless of the transmitted value (e.g. the TCP/IP stack in our implementation) are equally active during the two periods and are therefore not correlated to x.

The Receive process should probe each set in $\tilde{S}$ once for every x that the send process transmits. Probing at a slower rate wastes transmissions, while probing at a higher rate increases the noise since the number of times the receiver obtains data on the communication channel is at most the number of times that x is sent (each probe of a set s flushes that set until the next transmission). Therefore, the size of the input collection $|\tilde{S}| = p$ is chosen so that running prime and probe p times is at most $interval$ clock cycles.

3.3 Setup Phase

In order to initialize the communication channel, the attacker has to locate the cache set that stores array1_size, the value of array1_size and the cache sets that store array2[$256 * b$] for $b = 0, \ldots, 255$. The attacker begins by estimating the values of the $w, n, interval$ and α

Send(w, n, x, x')

(1) Do n times
  (a) $clock \leftarrow time()$
  (b) While $(time() - clock) < w$) do
    (i) Transmit x to the victim.
  (c) $clock \leftarrow time()$
  (d) While $(time() - clock) < w$) do
    (i) Transmit x' to the victim.

Receive($w, n, interval, \tilde{S}$)

(1) For each $s \in \tilde{S}$, Let $r_s \leftarrow 0, r'_s \leftarrow 0$
(2) Let $sendx \leftarrow$ True
(3) Do n times
  (a) Let $clock1 \leftarrow time()$
  (b) While $(time() - clock1 < w)$ do
    (i) Let $clock2 \leftarrow time()$
    (ii) For every $s \in \tilde{S}$ do
      (A) If $(sendx)$ then $r_s \leftarrow r_s + Probe(s)$
        Else then $r'_s \leftarrow r'_s + Probe(s)$
    (iii) While $(time() - clock2 < interval)$ Wait
  (c) $sendx \leftarrow !sendx$
  (d) If $(sendx)$ then //Every 2w
    (i) For each $s \in \tilde{S}$ do
      (A) if $(r_s > r'_s)$ then $score(s) \leftarrow score(s) + 1/n$
      (B) $r_s \leftarrow 0, r'_s \leftarrow 0$

Figure 3: The send and receive algorithms. *Probe*(s) **performs a cache probe of the cache set** s **and returns the number of cache misses.**

which are system dependent and then proceeds with the following steps.

3.3.1 Discovering the value of $\boldsymbol{array1\_size}$. Let S be the collection of *all* cache sets. For every $0 \le x$ the attacker computes a set $S'_x =$ LocateSets$(S, p, w, n, interval, x, -1, \alpha)$ until $S'_x = \emptyset$. The attacker then assigns $array1\_size = x$.

When the Send process transmits $x, 0 \le x < array1\_size$ to the victim then the if statement in Figure 1 is true, but when the victim receives -1 that statement is false. As a result S'_x contains the cache sets that store array2[array1[$x * 256$]] and array1[x]. If $x \ge array1\_size$ then S'_x is expected to be empty as the if statement in Figure 1 is false and the same sets will be active during the transmission of x and the transmission of -1.

3.3.2 Locating array1 and array2. Define S_{array2} to be the collection of cache sets that store array2[$256 * b$] for $0 \le b \le 255$. Begin by constructing upper and lower bounds on array2[$256 * b$], i.e. S_ℓ, S_u such that $S_\ell \subseteq$ array2[$256 * b$] $\subseteq S_u$ and locates the cache set that holds array1.

Initialize $S_\ell \leftarrow \bigcup_x S'_x$ and remove any cache set s that appears in successive collections $S'_x, \ldots, S'_{x+c}$ for $0 \le x < array1\_size - c$ and some small constant $c < B$. For every $0 \le x < array1\_size$ the set S'_x contains the cache set that stores array2[$256 *$ array1[x]], the cache set that stores array1[x] and possibly the cache set that stores array2. If n and w are sufficiently large in the LocateSets algorithm then S'_x includes only these three sets. Every contiguous B addresses of array1 are stored in a single cache line and therefore, except for the boundaries of the array, the same cache set appears in B successive collections $S'_x, \ldots, S'_{x+B-1}$. Therefore, the attacker learns the cache set that holds $array1$ and in addition $S_\ell \subseteq S_{array2}$. Furthermore, unless the values of $array1$ are arranged in runs of identical values that are all longer than c then $S_\ell \ne \emptyset$.

Compute S_u from S_ℓ by exploiting the contiguous layout of array2 in memory and the mapping of that portion of memory to the LLC. If a line is of length B bytes, there are L sets in a slice and the number of cores is a power of two then LB contiguous bytes in memory, beginning at a slice boundary, are stored in order in a single LLC slice. If the LB bytes begin in cache line i, which is not at the slice boundary then these are stored in at most two slices in sets $i, \ldots, L$ in the first slice and $1, \ldots, i-1$ in the second slice.

In all current Intel architectures we are aware of including the architectures we tested, $L \ge 1024$ and $B = 64$. Therefore, the 256 addresses of array2 are stored one address per cache line in cache sets that are exactly four apart and reside in at most two slices.

As a consequence, either all the cache sets of S_ℓ are in the same cache slice or they are part of at most two slices. In the first case, all the cache sets of S_ℓ are in slice C between cache sets a and b. In this case, if S_{array2} includes cache sets outside C they could potentially reside in any of the other slices. We introduce the following notation to define S_u. Denote the i-th cache set in slice C by $C[i]$, let $i_4 = i \bmod 4$ and let $i^4 = L - 4 + i_4$. In the first case, let

$$\begin{aligned} S_u &= \{C[a+4i] \mid 0 \le i < 256, a+4i < L\} \\ &\bigcup \{C'[a_4+4i] \mid \forall \text{ slice } C', 0 \le i < 256 - \lceil \frac{L-a}{4} \rceil\} \\ &\bigcup \{C[b-4i] \mid 0 \le i < 256, b-4i \ge 0\} \\ &\bigcup \{C'[b^4-4i] \mid \forall \text{ slice } C', 0 \le i < 255 - \lfloor b/4 \rfloor\} \end{aligned}$$

In the second case, S_ℓ is located in two slices C and C' between the sets a and b in C and a' and b' in C'. In this case the two slices of S_{array2} are already known and w.l.o.g we assume that $a > b'$ and the definition of S_u is

$$\begin{aligned} S_u &= \{C[a+4i] \mid 0 \le i < 256, a+4i < L\} \\ &\bigcup \{C[b-4i] \mid 0 \le i < 256 - \lceil \frac{b'}{4} \rceil - \lceil \frac{L-b}{4} \rceil\} \\ &\bigcup \{C'[a'+4i] \mid 0 \le i < 256 - \lceil \frac{a'}{4} \rceil - \lceil \frac{L-a}{4} \rceil\} \\ &\bigcup \{C[b'-4i] \mid 0 \le i < 256, b'-4i \ge 0\} \end{aligned}$$

3.3.3 array1_size. The next step is to detect the cache set that holds $array1\_size$. The main observation is that if $array1\_size$ is stored in the cache then speculative execution does not occur. We use the SendSE algorithm which is a modification of *Send* that during even numbered windows of w cycles sends only a legal x' value and during odd numbered windows it first sends two x' followed by an out-of-bounds x hoping to cause a misprediction. Consequently, the LocateSets method is also modified to receive and pass down to SendSE a set s_{size} which is a candidate for the set that holds $array1\_size$. For every set $s \in S$ the attacker tests whether it stores $array1\_size$ by setting $s_{size} \leftarrow s$ and running $S'_x =$ LocateSets$(S_{array2}, p, w, n, interval, x, x', \alpha, s)$ for some $x \ge array1\_size$ and $0 \le x' < array1\_size$. With overwhelming probability S'_x will not be empty and will contain the set that

stores array2[array1[x] $*$ 256] only if s is the cache set that stores $array1\_size$.

3.4 Reliable Transmission and Reception

After the setup phase, the attacker can try to mount a Spectre attack as follows. To learn an out-of-bounds address array1[x] the attacker can send several legal values x', prime the set s_{size} to remove array1_size from the cache, prime the cache sets of S_u, send x and then probe the cache sets of S_u to look for activity.

However, the attacker needs to overcome two problems with the straightforward Spectre attack: noise in the measurements and the difference between S_{array2} and S_u. The first source of noise is software activity that is independent of the attacker and that affects the cache and therefore cache measurements. Such activity is inevitable in a multi-tenant and multi-process system, but since it is independent of the attacker, it is likely that by repeating measurements sufficiently many times the activity in a cache set storing array2[array1[x] $*$ 256] will be on average noticeably higher than a cache set storing some independent instructions and data.

The second source of noise is due to failed synchronization between the send and receive processes of LocateSets. The main mitigation for this issue is to choose a large enough value w.

The third source of noise is triggered by the attacker itself in the process of running the attack. One problem can arise if one of the cache sets that stores part of the attacker's code that is used for transmitting x, but not for transmitting x', is also part of S_u. Such a cache set will be correlated with transmission of x, without being the correct set holding array2[array1[x] $*$ 256]. A second problem occurs if the cache set storing array1[x] happens to be one of the sets storing array2[256 $*$ b]. Since array1 is practically unbounded, this type of noise is very likely to occur for some values of x resulting in LocateSets returning several candidates for array2[array1[x] $*$ 256].

To counter the first type of attacker generated noise, the attacker performs memory accesses to the corresponding instructions regardless of their execution during the two periods, transmission of x (after transmitting x' twice) and transmission of x' only. The result of this idea is that the activity generated by the attacker's code will be exactly identical in both periods. In order to reduce the second type of attacker-generated noise, recall that after the setup phase, the attacker knows the cache set that holds array1. Moreover, due to the contiguous layout of array1 in memory, the attacker can predict the cache sets that store array1[i] for $i \geq 0$. The attacker performs a memory access to the cache set that stores array1[x] within each period that x is not transmitted, again to remove the correlation of this set to x.

If $S_{array2} \subsetneq S_u$ then the attack is slower, because all the sets of S_u must be checked for activity. In addition, correlating the activity of a set in array2 to a transmitted x does not necessarily reveal the value of array1[x] since the location of the set within array2 may not be clear. As the attacker proceeds, extracting correlated sets for values $x \geq$ array1_size, more values in array2 will be revealed, leading to a recalculation of S_ℓ and S_u. Even if a gap remains between the two collections the attack can still proceed. Note that the distance between active cache sets, i.e. the cache sets in S_ℓ, is known and the only possible missing information to determine each set correctly is the location of the first address of array2. Therefore, the values that attacker extracts act as the ciphertext of a shift cipher in which the possible shift is one of 256 values, which in practice will be easily broken by guessing the shift.

SendSE-I(w, n, x, x', s_{size})

(1) Do n times
  (a) $clock \leftarrow time()$
  (b) While ($time() - clock < w$) do
    (i) Repeat for $x_1 = x'$, $x_2 = x'$ and $x_3 = x$
      (A) Probe(s_{size}) // Flush the set storing $array1\_size$
      (B) Transmit x_i to the victim.
  (c) $clock \leftarrow time()$
  (d) While ($time() - clock < w$) do
    (i) Probe(s_{size})
    (ii) Transmit x' to the victim.
    (iii) Access addresses of instructions in 1b, which do not appear in 1(d)i and 1(d)ii.
    (iv) Access a memory address which is mapped to the same cache set as $array1[x]$.

Figure 4: Improved SendSE with the noise cancellation of Section 3.4

3.5 Performance Optimization

There are several ways to optimize the rate at which the attack extracts information from the victim.

3.5.1 Tuning parameters. An obvious goal is to tune the various parameters of the attack so that performance is optimized with minimal impact on reliability. Some of the parameters including $interval$, w and n depend heavily on system hardware, software and environment. For example, the size of a set in the LLC determines

ExractData($x_{init}, x_{term}, S, p, w, n, interval, x', \alpha, \beta, s_{size}$)

1. For each $x_{init} \leq x \leq x_{term}$ do
  (a) $S_{tmp} \leftarrow S$
  (b) For each $s \in S_{tmp}$ do $score(s) \leftarrow 0$
  (c) While ($\max\{score(s) \mid s \in S_{tmp}\} \leq \alpha$) do
    i. Partition S into subsets $\{S_1, S_2, \ldots, S_{|S_{tmp}|/p}\}$ of size at most p.
    ii. For every $\tilde{S} \in \{S_1, S_2, \ldots, S_{|S_{tmp}|/p}\}$
      A. Run simultaneously
        (I) SendSE-I(w, n, x, x', s_{size})
        (II) Receive($w, n, interval, \tilde{S}$)
    iii. For every $s \in S_{tmp}$ do
      A. If ($score(s) > \alpha$) then output (x, s).
      B. If ($score(s) < \beta$) then $S_{tmp} \leftarrow S_{tmp} \setminus \{s\}$.

Figure 5: Data extraction process. The algorithm receives as input a range of x values from x_{init} to x_{term}, a collection of sets S (typically $S = S_u$), architecture dependent parameters p, w, n and $interval$, a legal index x' for array1, the thresholds α and β and the set s_{size} that holds array1_size. The algorithm outputs for each x the corresponding set s which can be used to learn array1[x].

the minimum time for probing the set, while the round-trip-time between the sender and the target impacts *interval* and w. In Section 4 we provide details on the parameters we used for our tests.

An interesting parameter is the number of times that the branch predictor needs to receive input that enters a specific branch to be trained for that branch. Our tests confirm that in Intel's Haswell and Skylake architectures the branch predictor can be trained with two inputs. Therefore, we consistently send exactly two legal inputs x' followed by an out-of-bounds input x to trigger speculative execution.

3.5.2 Repeated noise. Common cache replacement algorithms use "pseudo"-LRU policies indicating that recently accessed memory addresses are likely to be accessed again in the near future and should therefore remain in the cache. When LocateSets runs SendSE and Receive in parallel, the same set s is repeatedly probed by Receive while SendSE repeatedly transmits an input x. While this strategy will read x as many times as it is sent (assuming that Prime+Probe is faster than transmission, which is typically the case), it will also read a lot of repeated "noise", i.e. memory lines which are mapped to the same cache set as array2[array1[x] $*$ 256] that are accessed due to some other process in the system. The benefit of probing a set s many times in succession is therefore much lower than accessing at longer intervals due to the temporal locality of most processes. Therefore, the value of *interval* should typically be significantly larger than the bare minimum required for sending an out-of bounds x and probing a set. Indeed, our approach is to set *interval* to be large enough to probe many sets (between ten and a hundred) within the same interval.

3.5.3 Non-uniform sampling of sets. The process described in LocateSets probes each cache set the same number of times, disregarding differences between the sets that can be translated to savings in time and communication. The goal of this optimization is to identify cache sets that are unlikely to hold array2[array1[x] $*$ 256] dynamically during the receive process. These sets are discarded from the collection of sets being probed, leaving more time for likelier sets to be sampled.

Let $t_x, t_{x'}$ be time windows of length w cycles each such that the send process transmits x during t_x and transmit x' during $t_{x'}$. If a set s holds array2[array1[x] $*$ 256] and is "quiet", i.e. has very little activity that is unrelated to x, then the number of cache misses during t_x is likely to be higher than the number of cache misses during $t_{x'}$. If s holds array2[array1[x] $*$ 256] and is "noisy", then this measurement may need to be repeated several times to reach the correct conclusion about s. Furthermore, if *s does not* hold array2[array1[x] $*$ 256] then it is very unlikely that over a sufficient number of measurements activity in t_x will be significantly higher than activity in $t_{x'}$.

To decide which sets are likely to hold array2[array1[x] $*$ 256]. We use a threshold n as the number of times the activity of a set is compared between periods t_x and $t_{x'}$ and two rating thresholds $0 \leq \beta < \alpha \leq 1$.

The heuristic rating of cache sets proceeds as follows. Initialize a collection of sets $S \leftarrow S_u$ and iterate over S probing each cache set $s \in S$ during several t_x and $t_{x'}$ windows. If after n iterations, $score(s) < \beta$ then remove s from S for the next iteration. If there exists a set $s \in S$ such that $score(s) > \alpha$ then return s as the likely set to hold array2[array1[x] $*$ 256].

3.6 Putting it all together

In this section we put together the insights from Sections 3.4 and 3.5 to improve the basic data extraction algorithms. Pseudo code for the improved SendSE process, SendSE-I appears in Figure 4. The Receive process remains the same as presented earlier in Figure 3.

Pseudo-code for the data extraction process appears in Figure 5. The algorithm attempts to extract every byte from array1[x] by using $SendSE-I$ and $Receive$ to grade the sets based on their cache misses sampled in $Receive$. The algorithm either discards them for the current x or returns a likely candidate for a cache set that stores array2[array1[x] $*$ 256].

4 EVALUATION

We evaluated the attack in a lab environment and report on the rate and the accuracy of the setup and attack phases. The laboratory setup included a Lenovo Ideapad Y700 laptop equipped with an Intel i7-6700HQ Processor (Skylake family), running Ubuntu 16.04.4 LTS, as well as a desktop PC with an Intel i5-4590 Processor (Haswell family), running Ubuntu 14.04 LTS. Both processors have four physical cores and an LLC of size six megabyte. However, the Skylake processor has hyper-threading, which enables eight virtual cores and therefore the LLC is divided into eight slices of 1024 sets each. The Haswell processor does not have hyper-threading and its LLC is divided into four slices of 2048 sets each.

On each machine, the attack was carried out in two virtualization environments, using either VMs or Linux containers. In the first case the VMs were compatible with x86 instructions, running Lubuntu 18.04. In each test a simple victim ran as a VM (or a container) on one core, while the attacker ran on two cores (physical cores in the Haswell and two virtual cores on a single physical core in the Skylake). One of the attacker's cores was used for the Send process and the other for executing the rest of the attack. The source code of the attack software we developed can be downloaded from [1]. The software uses the Mastik toolkit [23] for cache attacks.

4.1 Setup Phase

The attacker performs the setup in two main stages. In the first it discovers the sets S_ℓ, S_u and the value of array1_size. In the second stage the attacker finds the cache set that stores array1_size. The time required for the first stage is linear in the value of array1_size and the time required for the second stage is linear in the size of the LLC (no information on the location of array1_size is assumed). The evaluation optimizes the second stage compared to the description in Section 3.3 by using the non-uniform sampling optimization of Section 3.5. This optimization results in a complex relation between the running time of the second stage, the size of S_u and the noise in the system.

The w and n parameters were chosen for accuracy. Smaller values lead to more errors, while higher values lead to slower running time without improving accuracy. The value of interval was chosen to be much larger than the minimum (which was about 4000 clock cycles for containers and 20000 cycles for VMs) to reduce the effects of system noise. In all measurements, $\alpha = 0.9$ and $\beta = 0.55$.

	Stage 1 time (seconds)	Stage 2 worst case time (seconds)	Stage 2 best case time (seconds)	Parameter choice		
				w (cycles)	n	interval(cycles)
Haswell VMs	$5.9 * array1\_size$	2191	1170	$5 \cdot 10^6$	16	$2 \cdot 10^5$
Haswell Containers	$1.17 * array1\_size$	456	192	$2 \cdot 10^6$	10	$2 \cdot 10^5$
Skylake VMs	$2.16 * array1\_size$	3503	732	$5 \cdot 10^6$	10	$2 \cdot 10^5$
Skylake Containers	$1.15 * array1\_size$	837	242	$2 \cdot 10^6$	10	$2 \cdot 10^5$

Table 1: Time measurements for the setup phase. Stage 2 worst case refers to the largest possible size for S_u, which is 255 or 256 sets in each slice, resulting from $|S_\ell| = 1$. Stage 2 best case refers to $S_u = S_{array2}$, i.e. 256 sets in a single slice.

	Accuracy		Rate		Stress (Accuracy)		Stress (Rate)	
	Prob.	Bytes/sec	Prob.	Bytes/sec	Prob.	Bytes/sec	Prob.	Bytes/sec
Haswell VMs	0.95	1.52	0.93	1.78	0.98	0.41	0.91	0.90
Haswell Containers	0.98	5.84	0.96	9.34	0.97	4.92	0.94	9.09
Skylake VMs	0.99	2.93	0.96	6.71	0.96	0.84	0.90	1.33
Skylake Containers	0.96	10.0	0.96	10.0	0.96	3.33	0.96	3.33

Table 2: Attack Performance: Each entry includes the probability of successful byte extraction and the extraction rate. Parameters are tuned for each case and range between $w = 2 \cdot 10^6, n = 7, interval = 5 \cdot 10^4$ for containers to $w = 10^7, n = 60$ and $interval = 4 \cdot 10^5$ for VMs. The thresholds are $\alpha = 0.90$ and $\beta = 0.55$.

4.2 Attack Phase

We use two measures to evaluate the attack, its *rate*, i.e. the number of bytes per second that the attack retrieves, and its *accuracy*, i.e. the probability that a byte that the attack extracts is correct.

We perform the evaluation for both hardware platforms and virtualization environments in four different regimes. We run the tests on either a relatively quiet system or using a stress tool [11] that runs at maximum utilization to allocate large memory regions, thus causing sustained activity in the LLC. In addition, we tune parameters either to a regime of maximum accuracy, defined as probability of correctly extracting each byte in the range $0.95 - 1$ per byte, or for higher rate while maintaining at least 0.9 probability of success per byte. The results are summarized in Table 2.

5 MONITOR AND STEALTH ATTACK

5.1 Monitoring Architecture Design Scheme

5.2 Stealth Attack

Since the attack as described can be detected, we wish to reduce the impact the attack has on the various hardware counters such as L3_TCM and BR_MSP. It is clear that in order to lower these counters from the victim's perspective, the attacker needs to redesign some of the algorithms described as to lower the intensity of branch mispredictions and LLC misses. First, the attacker needs to apply random idle intervals frequently as possible in order to not pose as a suspect the majority of the time. Second, the attacker, when not in idle mode, needs to perform actions such that only attack-crucial (atomic) bundle of instructions are not separated by idle intervals.

REFERENCES

[1] 2018. Spectre Without Shared Memory. https://github.com/amosbe/spectre-without-shared-memory.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 312–320.

[3] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441* (2018).

[5] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 368–379.

[6] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–Bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 490–505.

[7] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 191–205.

[8] Intel. 2018. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[9] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 299–319.

[10] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203* (2018).

[11] Linux. Accessed 2018. Stress - Load and Stress Test Tool. https://linux.die.net/man/1/stress.

[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[13] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 605–622.

[14] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1406–1418.

[15] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

[16] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 199–212.

[17] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv preprint arXiv:1802.03802* (2018).

[18] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 62–76.

[19] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels.. In *USENIX Security Symposium*. 687–702.

[20] VMware. 2018. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735). https://kb.vmware.com/s/article/2080735.

[21] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.. In *USENIX Security symposium*. 159–173.

[22] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 29–40.

[23] Yuval Yarom. 2016. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs. adelaide. edu. au/˜ yval/Mastik* (2016).

[24] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*. 719–732.