NetSpectre: Read Arbitrary Memory over Network

Michael Schwarz[1], Martin Schwarzl[1], Moritz Lipp[1], Jon Masters[2], and Daniel Gruss[1]

[1] Graz University of Technology, Austria
[2] Red Hat, United States

Abstract. All Spectre attacks so far required local code execution. We present the first fully remote Spectre attack. For this purpose, we demonstrate the first access-driven remote Evict+Reload cache attack over the network, leaking 15 bits per hour. We present a novel high-performance AVX-based covert channel that we use in our cache-free Spectre attack. We show that in particular remote Spectre attacks perform significantly better with the AVX-based covert channel, leaking 60 bits per hour from the target system. We demonstrate practical NetSpectre attacks on the Google cloud, remotely leaking data and remotely breaking ASLR.

1 Introduction

Over the past 20 years, software-based microarchitectural attacks have evolved from theoretical attacks [36] on implementations of cryptographic algorithms [49], to more generic practical attacks [60, 25], and recently to high potential threats [38, 35, 55, 47, 58] breaking the fundamental memory and process isolation. Spectre [35] is a microarchitectural attack, tricking another program into speculatively executing an instruction sequence which leaves microarchitectural side effects. Except for SMoTherSpectre [10], all Spectre attacks demonstrated so far [12] exploit timing differences caused by the pollution of data caches.

By manipulating the branch prediction, Spectre tricks a process into performing a sequence of memory accesses which leak secrets from chosen virtual memory locations to the attacker. Spectre attacks have so far been demonstrated in JavaScript [35] and native code [35, 59, 14, 41, 37, 27], but it is likely that any environment allowing sufficiently accurate timing measurements and some form of code execution enables these attacks. Attacks on Intel SGX enclaves showed that enclaves are also vulnerable to Spectre attacks [14]. However, there are many devices which never run any attacker-controlled code, *i.e.*, no JavaScript, no native code, and no other form of code execution on the target system. Until now, these systems were believed to be safe against such attacks. In fact, while some vendors discuss remote targets [8, 43] others are convinced that these systems are still safe and recommend to not take any action on these devices [32].

In this paper, we present NetSpectre, a new attack based on Spectre, requiring no attacker-controlled code on the target device, thus affecting billions of devices. Similar to a local Spectre attack, our remote attack requires the presence of a Spectre gadget in the code of the target. We show that systems containing the required Spectre gadgets in an exposed network interface or API can be attacked with our generic remote Spectre attack, allowing to read arbitrary memory over the network. The attacker only sends a series of requests and measures the response time to leak a secret from the victim.

We show that memory access latency, in general, is reflected in the latency of network requests. Hence, we demonstrate that it is possible for an attacker to distinguish cache hits and misses on specific cache lines remotely, by measuring and averaging over a larger number of measurements (law of large numbers). Based on this, we implemented the first access-driven remote cache attack, a remote variant of Evict+Reload called *Thrash+Reload*. We facilitate this technique to retrofit existing Spectre attacks to a network-based scenario and leak 15 bits per hour from a vulnerable target system.

By using a novel side channel based on the execution time of AVX2 instructions, we demonstrate the first Spectre attack which does not rely on a cache covert channel. Our AVX-based covert channel achieves a native code performance of 125 bytes per second at an error rate of 0.58 %. This covert channel achieves a higher performance in our NetSpectre attack than the cache covert channel. As cache eviction is not necessary anymore, we increase the speed to leaking 60 bits per hour from the target system in a local area network. In the Google cloud, we leak around 3 bits per hour from another virtual machine (VM).

We demonstrate that using previously ignored gadgets allows breaking address-space layout randomization in a remote attack. Address-space layout randomization (ASLR) is a defense mechanism deployed on most systems today, randomizing virtually all addresses. An attacker with local code execution can easily bypass ASLR since ASLR mostly aims at defending against remote attacks but not local attacks. Hence, many weaker gadgets for Spectre attacks were ignored so far, since they do not allow leaking actual data, but only address information. However, in the remote attack scenario weaker gadgets are still very powerful.

Spectre gadgets can be more versatile than anticipated in previous work. This not only becomes apparent with the weaker gadgets we use in our remote ASLR break but even more so with the value-thresholding technique we propose. Value-thresholding leaks bit-by-bit by through comparisons, by using a divide-and-conquer approach similar to a binary search.

Contributions. The contributions of this work are:
1. We present the first access-driven remote cache attack (Evict+Reload) and the first remote Spectre attack.
2. We demonstrate the first Spectre attack which does not use the cache but a new and fast AVX-based covert channel.
3. We use simpler Spectre gadgets in remote ASLR breaks.

Outline. Section 2 provides background. Section 3 overviews NetSpectre. Section 4 presents new remote covert channels. Section 5 details our attack. Section 6 evaluates the performance of NetSpectre. We conclude in Section 7.

2 Background

Modern CPUs have multiple execution units operating in parallel and precomputing results. To retain the architecturally defined execution order, a reorder buffer stores results until they are ready to be retired (made visible on the architectural level) in the order defined by the instruction stream. To keep precomputing, predictions are often necessary using e.g., on branch prediction. To optimize the prediction quality, modern CPUs incorporate several branch prediction mechanisms. If an interrupt occurs or a misprediction is unrolled, any precomputed results are architecturally discarded, however, the microarchitectural state is not reverted. Executed instructions that are not retired are called transient instructions [35, 38, 12].

Microarchitectural side-channel attacks exploit different microarchitectural elements. They were first explored for attacks on cryptographic algorithms [36, 49, 60] but today are generic attack techniques for a wide range of attack targets. Cache attacks exploit timing differences introduced by small in-CPU memory buffers. Different cache attack techniques have been proposed in the past, including Prime+Probe [49, 52], and Flush+Reload [60]. In a covert channel, the attacker controls both, the part that induces the side effect, and the part that measures the side effect. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [39, 45, 24].

Meltdown [38] and Spectre [35] use covert channels to transmit data from the transient execution to a persistent state. Meltdown exploits vulnerable deferred permission checks. Spectre [35] exploits speculative execution in general. Hence, they do not rely on any vulnerability, but solely on optimizations. Through manipulation of the branch prediction mechanisms, an attacker lures a victim process into executing attacker-chosen code gadgets. This enables the attacker to establish a covert channel from the speculative execution in the victim process to a receiver process under attacker control.

SIMD (single instruction multiple data) instructions enable parallel operation on multiple data values. They are available as instruction set extensions on modern CPUs, e.g., Intel MMX [29, 28, 30, 51], AMD 3DNow! [4, 48], and ARM VFP and NEON [7, 6, 3]. On Intel, some of the SIMD instructions are processed by a dedicated SIMD unit within the CPU core. However, to save energy, the SIMD unit is turned off when not used. Consequently, to execute such instructions, the SIMD unit is first powered up, introducing a small latency on the first few instructions [18]. Liu [40] noted that some SIMD instructions can be used to improve bus-contention covert channels. However, so far, SIMD instructions have not yet been used for pure SIMD covert channels or side-channel attacks.

One security mechanism present in modern operating systems is address-space layout randomization (ASLR) [50]. It randomizes the locations of objects or regions in memory, e.g., heap objects and stacks, so that an attacker cannot predict correct addresses. Naturally, this is a probabilistic approach, but it provides a significant gain in security in practice. ASLR especially aims at mitigating control-flow-hijacking attacks, but it also makes other remote attacks difficult where the attacker has to provide a specific address.

3 Attack Overview

The building blocks of a NetSpectre attack are two *NetSpectre gadgets*: a *leak gadget*, and a *transmit gadget*. We discuss the roles of these gadgets, which allow an attacker to perform a Spectre attack without any local code execution or access, based on their type (leak or transmit) and the microarchitectural element they use (e.g., cache).

Spectre attacks induce a victim to speculatively perform operations that do not occur in strict in-order processing of the program's instructions, and which leak a victim's confidential information via a covert channel to an attacker. Multiple Spectre variants are exploiting different prediction mechanisms. Spectre-PHT (also known as Variant 1) [35, 34] mistrains a conditional branch, e.g., a bounds check. Spectre-BTB (also known as Variant 2) [35] exploits mispredictions of indirect calls, Spectre-STL (also known as Variant 4) speculatively bypasses stores [27], and Spectre-RSB misuses the return stack buffer [37, 41]. While attack works with any Spectre variant, we focus on Spectre-PHT as it is widespread, illustrative, and difficult to fix in hardware [31, 12].

Before the value of a branch condition is known (resolved), the CPU predicts the most likely outcome and then continues with the corresponding code path. There are several reasons why the result of the condition is not known at the time of evaluation, e.g., a cache miss on parts of the condition, complex dependencies which are not yet satisfied, or a bottleneck in a required execution unit. By hiding these latencies, speculative execution leads to faster overall execution if the branch condition was predicted correctly. Intermediate results of a wrongly predicted condition are simply not committed to the architectural state, and the effective performance is similar to that which would have occurred had the CPU never performed any speculative execution. However, any modifications of the microarchitectural state that occurred during speculative execution, such as the cache state, are not reverted.

As our NetSpectre attack is mounted over the network, the victim device requires a network interface an attacker can reach. While this need not necessarily be Ethernet, a wireless or cellular link are also possible. Moreover, the target of the attack could also be baseband firmware running within a phone [8, 5]. The attacker must be able to send a large number of network packets to the victim but not necessarily within a short time frame. Furthermore, the content of the packets in our attack is not required to be attacker-controlled.

In contrast to local Spectre attacks, our NetSpectre attack is not split into two phases. Instead, the attacker constantly performs operations to mistrain the CPU, which will make it constantly run into exploitably erroneous speculative execution. NetSpectre does not mistrain across process boundaries, but instead trains in-place by passing in-bounds and out-of-bounds values alternatingly to the exposed interface.For our NetSpectre attack, the attacker requires two Spectre gadgets, which are executed if a network packet is received: a *leak gadget*, and a *transmit gadget*. The *leak gadget* accesses an array offset at an attacker-controlled index, compares it with a user provided value, and changes some microarchitectural state depending on the result of the comparison. The

```
if (x < length)
    if(array[x] > y)
        flag &= true
```

Listing 1.1: Excerpt of a function executed when a network packet is processed.

transmit gadget performs an arbitrary operation where the runtime depends on the microarchitectural state modified by the *leak gadget*. Hidden in a significant amount of noise, this timing difference can be observed in the network packet response time. Spectre gadgets can be found in modern network drivers, network stacks, and network service implementations.

To illustrate the working principle of our NetSpectre attack, we consider a basic example similar to the original Spectre-PHT example [35] in an adapted scenario: the code in Listing 1.1 is part of a function that is executed when a network packet is received. Note that this just one variant to enable bit-wise leakage, there is an abundance of other gadgets that leak a single bit. We assume that `x` is attacker-controlled, e.g., a field in a packet header or an index for some API. This code forms our *leak gadget*.

The code fragment begins with a bound check on `x`, a best practice for secure software. The attacker can remotely exploit speculative execution as follows:

1. The attacker sends multiple network packets with the value of `x` always in bounds. This trains the branch predictor, increasing the chance that the outcome of the comparison is predicted as true.
2. A packet where `x` is out of bounds is sent, such that `array[x]` is a secret value in the target's memory. However, the branch predictor still assumes the bounds check to be true, and the memory access is speculatively executed.
3. If the attacker-controlled value `y` is less than the secret value `array[x]`, the `flag` variable is accessed.

While changes are not committed architecturally after the condition is resolved, microarchitectural state changes are not reverted. Thus, in Listing 1.1, the cache state of `flag` changes although the value of `flag` does not change. Only if the attacker guessed `y` such that it is less than `array[x]`, `flag` is cached. Note that the operation on `flag` is not relevant as long as `flag` is accessed.

The *transmit gadget* is much simpler, as it only has to use `flag` in an arbitrary operation. Consequently, the execution time of the gadget will depend on the cache state of `flag`. In the most simple case, the *transmit gadget* simply returns the value of `flag`, which is set by the *leak gadget*. As the architectural state of `flag` (*i.e.*, its value) does not change for out-of-bounds `x`, it does not leak secret information. However, the response time of the *transmit gadget* depends on the microarchitectural state of `flag` (*i.e.*, whether it is cached), which leaks one secret bit of information.

To complete the attack, the attacker performs a binary search over the value range. Each tested value leaks one secret bit. As the difference in the response time is in the range of nanoseconds, the attacker needs to average over a large number of measurements to obtain the secret value with acceptable confidence.

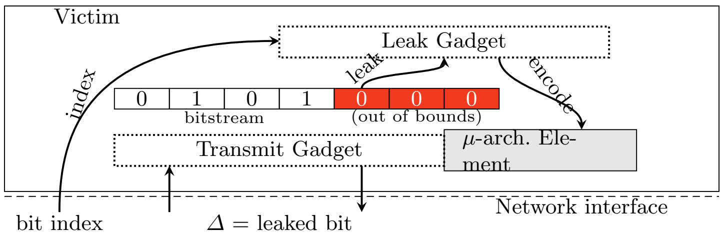

Fig. 1: The interaction of the *NetSpectre gadget* types.

Indeed, our experiments show that the difference in the microarchitectural state becomes visible when performing a large number of measurements. Hence, an attacker can first measure the two corner cases (*i.e.*, cached and uncached) and afterward, to extract a real secret bit, perform as many measurements as necessary to distinguish which case it is with confidence, e.g., using a threshold or a Bayes classifier.

We refer to the two gadgets, the *leak gadget* and the *transmit gadget*, as *NetSpectre gadgets*. Running a *NetSpectre gadget* may require sending more than one packet. Furthermore, the *leak gadget* and *transmit gadget* may be reachable via different independent interfaces, *i.e.*, both interfaces must be attacker-accessible. Figure 1 illustrates the two gadgets types that are detailed in Section 3.2.

From the listings illustrating gadgets, it is clear that such code snippets exist in real-world code (cf. Listing 1.3). However, as they can potentially be spread across many instructions and might not be visible in the source code, identifying such gadgets is currently an open problem which is also discussed in other Spectre papers [35, 34, 37, 41]. Moreover, the reachability of a gadget with specific constraints is an orthogonal problem and out of scope for this paper. As a consequence, we follow best practices by introducing Spectre gadgets into software run by the victim to evaluate the attack in the same manner as other Spectre papers [34, 37, 41]. Suitable gadgets can be located in real-world software applications through static analysis of source code or through binary inspection.

3.1 Gadget location

The set of attack targets depends on the location of the *NetSpectre gadgets*. As illustrated in Figure 2, on a high level, there are two different gadget locations:

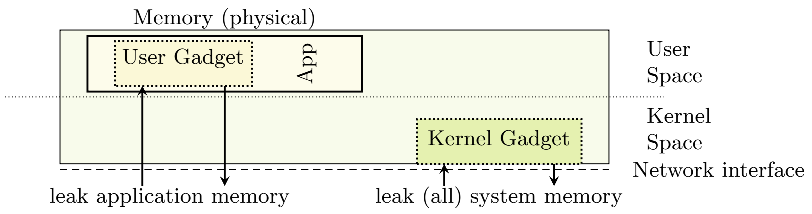

Fig. 2: Depending on the gadget location, the attacker can access memory of the application or the entire kernel, typically including all system memory.

in the user space or in the kernel space. However, they can also be found in software running below, e.g., hypervisor, baseband or firmware.

Attacks on the Kernel. The network driver is usually implemented in the kernel of the operating system, either as a fixed component or as a kernel module. In either case, kernel code is executed when a network packet is received. If any kernel code processed during the handling of the network packet contains a *NetSpectre gadget*, *i.e.*, an attacker-controlled part of the packet is used as an index, comparing the array value with a second user-controlled value, a NetSpectre attack is possible.

An attack on the kernel code is particularly powerful, as the kernel does not only have the kernel memory mapped but typically also the entire physical memory. On Linux and macOS, the physical memory can be accessed via the direct-physical map, *i.e.*, every physical memory location is accessible via a predefined virtual address in the kernel address space. Windows does not use a direct-physical map but maintains memory pools, which typically also map a large fraction of the physical memory. Thus, a NetSpectre attack using a *NetSpectre gadget* in the kernel can in general leak arbitrary values from memory.

Attacks on the User Space. Usually, network packets are not only handled by the kernel but are passed on to a user-space application which processes the content of the packet. Hence, not only the kernel but also user-space applications can contain *NetSpectre gadgets*. In fact, all code paths that are executed when a network packet arrives are candidates to look for *NetSpectre gadgets*. This does include code both on the server side and the client side.

An advantage in attacking user-space applications is the significantly larger attack surface, as many applications process network packets. Especially on servers, there are an abundance of services processing user-controlled network packets, e.g., web servers, FTP servers, or SSH daemons. Moreover, a remote server can also attack a client machine, e.g., via web sockets, or SSH connections. In contrast to attacks on the kernel space, which in general can leak any data stored in the system memory, attacks on a user-space application can only leak secrets of the attacked application.

Such application-specific secrets include secrets of the application itself, e.g., credentials and keys. Thus, a NetSpectre attack using a *NetSpectre gadget* in an application can access arbitrary data processed by the application. Furthermore, if the victim is a multi-user application, e.g., a web server, it also contains the secrets of multiple users. This is especially interesting for popular websites with many users.

3.2 Gadget type

We now discuss the different *NetSpectre gadgets*; the *leak gadget* to encode a secret bit into a microarchitectural state, and the *transmit gadget* to transfer the microarchitectural state to a remote attacker.

Leak Gadget. A *leak gadget* leaks secret data by changing a microarchitectural state depending on the value of a memory location that is not directly accessible to the attacker. The state changes on the victim device, not directly observable over the network. A NetSpectre *leak gadget* only leaks a single bit. Single-bit gadgets are the most versatile, as storing a one-bit (binary) state can be accomplished with many microarchitectural states, as only two cases have to be distinguished (cf. Section 4). Thus, we focus on single-bit *leak gadgets* in this paper as they can be as simple as shown in Listing 1.1. In this example, a value (`flag`) is cached if the value at the attacker-chosen location is larger than the attacker-chosen value `y`. The attacker can use this gadget to leak secret bits into the microarchitectural state.

Transmit Gadget. In contrast to Spectre, NetSpectre requires an additional gadget to transmit the leaked information to the attacker. As the attacker does not control any code on the victim device, the recovery process, *i.e.*, observing the microarchitectural state, cannot be implemented by the attacker. Furthermore, the architectural state can usually not be accessed via the network and, thus, it would not even help if the gadget converts the state.

From the attacker's perspective, the microarchitectural state must become visible over the network. This may not only happen directly via the content of a network packet but also via side effects. Indeed, the microarchitectural state will in some cases become visible, e.g., in the form of the response time. We refer to a code fragment which exposes the microarchitectural state to a network-based attacker and which can be triggered by an attacker, as a *transmit gadget*. Naturally, the *transmit gadget* has to be located on the victim device. With a *transmit gadget*, the microarchitectural state measurement happens on a remote machine but exposes the microarchitectural state over a network-reachable interface.

In the original Spectre attack, Flush+Reload is used to transfer the microarchitectural state to an architectural state, which is then read by the attacker to leak the secret. The ideal case would be if such a Flush+Reload gadget is available on the victim, and the architectural state can be observed over the network. However, as it is unlikely to locate an exploitable Flush+Reload gadget on the victim and access the architectural state, regular Spectre gadgets cannot simply be retrofitted to mount a NetSpectre attack.

In the most direct case, the microarchitectural state becomes visible for a remote attacker, through the latency of a network packet. A simple *transmit gadget* for the *leak gadget* shown in Listing 1.1 just accesses the variable `flag`. The response time of the network packet depends on the cache state of the variable, *i.e.*, if the variable was accessed, the response takes less time. Generally, an attacker can observe changes in the microarchitectural state if such differences are measurable via the network.

4 Remote Microarchitectural Covert Channels

A cornerstone of our NetSpectre attack is building a microarchitectural covert channel that exposes information to a remote attacker (cf. Section 3). Since in our scenario the attacker cannot run any code on the target system, we use a *transmit gadget* whose execution can be triggered by the attacker. In this section, we present the first remote access-driven cache attack, *Thrash+Reload*, a variant

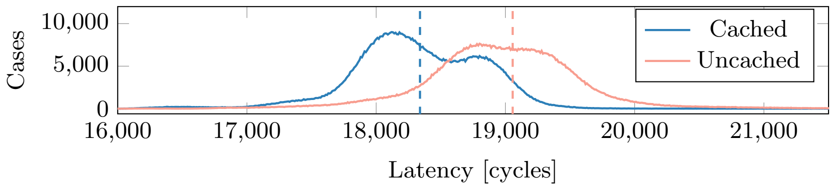

Fig. 3: Measuring the response time of a *transmit gadget* accessing a certain variable. Only by performing a large number of measurements, the difference in the response timings depending on the cache state becomes visible. The distribution's average values are shown as dashed lines.

of Evict+Reload. We show that with *Thrash+Reload*, an attacker can build a covert channel from the speculative execution on the target device to a remote receiving end on the attacker's machine. Furthermore, we also present a previously unknown microarchitectural covert channel based on AVX2 instructions. We show that this covert channel can be used in NetSpectre attacks, yielding even higher transmission rates than the remote cache covert channel.

4.1 Remote Cache Covert Channel

Kocher et al. [35] use the cache as the microarchitectural element to encode the leaked data. This allows using well-known cache side-channel attacks, such as Flush+Reload [60] or Prime+Probe [49, 52] to deduce the microarchitectural state and thus the encoded data. However, not only caches keep microarchitectural states which can be used for covert channels [53, 16, 11, 19, 56].

Mounting a Spectre attack by using the cache has three main advantages: there are powerful methods to make the cache state visible, many operations modify the cache state and are thus visible in the cache, and the timing difference between a cache hit and cache miss is comparably large. Flush+Reload is usually considered the most fine-grained and accurate cache attack, with almost zero noise [60, 24, 19]. If shared memory is not available, Prime+Probe is considered the next best choice [45, 57]. Consequently, all Spectre attacks published so far use either Flush+Reload [35, 14] or Prime+Probe [59].

For the first NetSpectre attack, we need to adapt local cache covert channel techniques. Instead of measuring the memory access time directly, we measure the response time of a network request which uses the corresponding memory location. Hence, the response time is influenced by the cache state of the variable used for the attack. The difference in the response time due to the cache state is in the range of nanoseconds since memory accesses are comparably fast.

The network latency is subject to many factors, leading to noisy results. However, the law of large numbers applies: no matter how much statistically independent noise is included, averaging over a large number reveals the signal [1, 33, 2, 9, 61]. Hence, an attacker can still obtain the secret value with confidence.

Figure 3 shows that the difference in the microarchitectural state is indeed visible when performing a large number of measurements. The average values

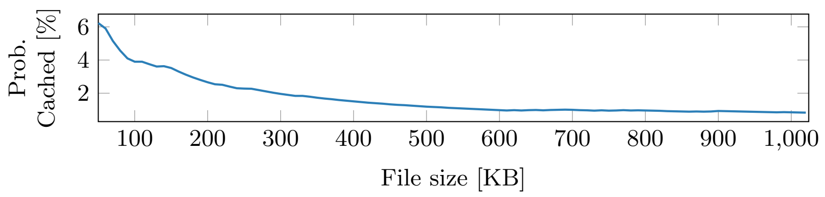

Fig. 4: The probability that a specific variable is evicted from the victim's last-level cache by downloading a file from the victim (Intel i5-6200U). The larger the downloaded file, the higher the probability that the variable is evicted.

of the two distributions are illustrated as dashed vertical lines. An attacker can either use a classifier on the measured values, or first measure the two corner cases (cached and uncached) to get a threshold for the real measurements.

Still, as the measurement destroys the cache state, *i.e.*, the variable is always cached after the first measurement, the attacker requires a method to evict (or flush) the variable from the cache. As it is unlikely that the victim provides an interface to flush or evict a variable directly, the attacker cannot use well-known cache attacks but has to resort to more crude methods. Instead of the targeted eviction in Evict+Reload, we simply evict the entire last-level cache by thrashing the cache, similar to Maurice et al. [44]. Hence, we call this technique *Thrash+Reload*. To thrash the entire cache without code execution, we use a network-accessible interface. In the simplest form, any packet sent from the victim to the attacker, e.g., a file download, can evict a variable from the cache.

Figure 4 shows the probability of evicting a specific variable (*i.e.*, the `flag` variable) from the last-level cache by requesting a file from the victim. The victim is running on an Intel i5-6200U with 3 MB last-level cache. Downloading a 590 kilobytes file evicts our variable with a probability of $\geq 99\,\%$.

With a mechanism to distinguish hits and misses, and a mechanism to evict the cache, we have all building blocks required for a cache side-channel attack or a cache covert channel. *Thrash+Reload* combines both mechanisms over a network interface, forming the first remote cache covert channel. In our experiments on a local area network, we achieve a transmission rate of up to 4 bit per minute, with an error rate of $< 0.1\,\%$. This is significantly slower than cache covert channels in a local native environment, e.g., the most similar attack (Evict+Reload) achieves a performance of 13.6 kb/s with an error rate of 3.79 %.

We use our remote cache covert channel for remote Spectre attacks. However, remote cache covert channels and especially remote cache side-channel attacks are an interesting object of study. Many attacks that were presented previously would be devastating if mounted over a network interface [60, 25, 22].

4.2 Remote AVX-based Covert Channel

To demonstrate the first Spectre variant which does not rely on the cache as the microarchitectural element, we require a covert channel which allows transmitting information from speculative execution to an architectural state. Thus, we

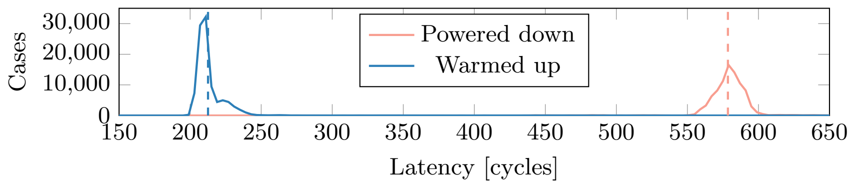

Fig. 5: If the AVX2 unit is inactive (powered down), executing an AVX2 instruction takes on average 366 cycles longer than on an active AVX2 unit (Intel i5-6200U). Average values shown as dashed lines.

build a novel covert channel based on timing differences in AVX2 instructions. This covert channel has a low error rate and high performance, and it allows for a significant performance improvement in our NetSpectre attack as compared to the remote cache covert channel.

To save power, the CPU can power down the upper half of the AVX2 unit which is used to perform operations on 256-bit registers. The upper half of the unit is powered up as soon as an instruction is executed which uses 256-bit values [46]. If it is not used for more than 1 ms, it is powered down [17].

Performing a 256-bit operation when the upper half is powered down incurs a significant performance penalty. For example, we measured the execution (including measurement overhead) of a simple bit-wise AND of two 256-bit registers (`VPAND`) on an Intel i5-6200U (cf. Figure 5). If the upper half is active, the operation takes on average 210 cycles, whereas if the upper half is powered down (*i.e.*, it is inactive), the operation takes on average 576 cycles. The difference of 366 cycles is even larger than the difference between cache hits and misses, which is only 160 cycles on the same system. Hence, the timing difference in AVX2 instructions is better for remote microarchitectural attacks.

Similarly to the cache, reading the latency of an AVX2 instruction also destroys the encoded information. Therefore, an attacker requires a method to reset the AVX2 unit, *i.e.*, power down the upper half. In contrast to the cache, this is easier, as the upper half of the AVX2 unit is automatically powered down after 1 ms of inactivity. Thus, an attacker only has to wait at least 1 ms.

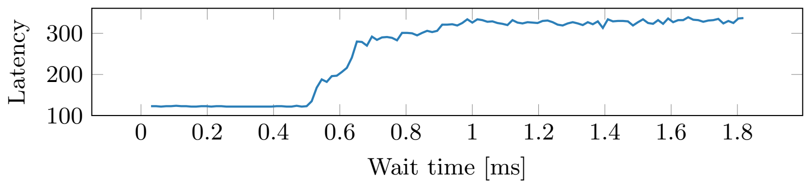

Fig. 6: The number of cycles it takes to execute the `VPAND` instruction (with measurement overhead) after not using the AVX2 unit. After 0.5 ms, the upper half of the AVX2 unit powers down, which increases the latency for subsequent AVX2 instructions. After 1 ms, it is fully powered down, and we see the maximum latency for subsequent AVX2 instructions.

```
if (x < length)
    if(array[x] > y)
        _mm256_instruction();
```

Listing 1.2: AVX2 NetSpectre gadget which encodes one bit of information.

Figure 6 shows the execution time of an AVX2 instruction (specifically `VPAND`) after inactivity of the AVX2 unit. If the inactivity is shorter than 0.5 ms, *i.e.*, the last AVX2 instruction was executed not more than 0.5 ms ago, there is no performance penalty when executing an AVX2 instruction which uses the upper half of the AVX2 unit. After that, the AVX2 unit begins powering down, increasing the execution time for any subsequent AVX2 instruction, as the unit has to be powered up again while emulating AVX2 in the meantime [17]. It is fully powered down after approximately 1 ms, leading to the highest performance penalty if any AVX2 instruction is executed in this state.

A *leak gadget* using AVX2 is similar to a *leak gadget* using the cache. Listing 1.2 shows (pseudo-)code of an AVX2 *leak gadget*. The `_mm256_instruction` represents an arbitrary 256-bit AVX2 instruction, e.g., `_mm256_and_si256`. If the referenced element `x` is larger than the user-controlled value `y`, the instruction is executed, and as a consequence, the upper half of the AVX2 unit is powered on. The power up also happens if the branch-prediction outcome of the bounds check was incorrect and the AVX2 instruction is accessed speculatively. Note that there is no data dependency between the AVX2 instruction and the array lookup. Only the information whether an AVX2 instruction was executed is used to transmit the secret bit of information through the covert channel.

The *transmit gadget* is again similar to the *transmit gadget* for the cache. Any function that uses an AVX2 instruction, and has thus a measurable runtime difference observable over the network, can be used as a *transmit gadget*. Even the *leak gadget* shown in Listing 1.2 can act as a *transmit gadget*. By providing an in-bounds value for `x`, the runtime of the function depends on the state of the upper half of the AVX2 unit. If the upper half of the unit was used before, *i.e.*, a '1'-bit (`array[x] > y`) was leaked, the function executes faster than if the upper half was not used before, *i.e.*, a '0'-bit (`array[x] <= y`) was leaked.

With these building blocks, we build the first pure-AVX covert channel and the first AVX-based remote covert channel. In our experiments in a native local environment, we achieve a transmission rate of 125 B/s with an error rate of 0.58 %. In a local area network, we achieve a transmission rate of 8 B/min, with an error rate of <0.1 %. Since the true capacity of this remote covert channel is higher than the true capacity of our remote cache covert channel, it yields higher performance in our NetSpectre attack.

5 Attack Variants

In this section, we first describe an attack to extract secret data via value-thresholding bit-by-bit from the memory of the target system. We then describe how to defeat ASLR on the remote machine, paving the way for remote exploitation. We use gadgets based on Spectre-PHT for illustrative purposes, but this can naturally be done with any Spectre gadget that lies in a code path reached from handling a remote packet.

5.1 Extracting Data from the Target

With typical *NetSpectre gadgets* (cf. Section 3), the attack consists of 4 steps. Depending on the gadgets, the *leak gadget* and *transmit gadget* can be the same.

1. Mistrain the branch predictor.
2. Reset the state of the microarchitectural element.
3. Leak a bit via value-thresholding to the microarchitectural element.
4. Expose the element state to the network.

In step 1, the attacker mistrains the branch predictor of the victim to run a Spectre attack by using the *leak gadget* with valid indices. The valid indices ensure that the branch predictor learns always to take the branch, *i.e.*, speculating that the condition is true. With no feedback to the attacker, the microarchitectural state does not have to be reset or transmitted.

In step 2, the attacker resets the microarchitectural state to enable encoding leaked bits using a microarchitectural element. This step depends on the used microarchitectural element, e.g., when using the cache, the attacker downloads a large file from the victim; for AVX2, the attacker waits for about 1 ms.

In step 3, the attacker exploits Spectre to leak a single bit from the victim. As the branch predictor is mistrained in step 1, providing an out-of-bounds index to the *leak gadget* will run the in-bounds path and modify the microarchitectural element, *i.e.*, the bit is encoded in the microarchitectural element.

In step 4, the attacker has to transmit the encoded information via the network. This step corresponds to the second phase of the original Spectre attack. In contrast to the original Spectre attack, which uses a cache attack, the attacker uses the *transmit gadget* for this step as described in Section 4. The attacker sends a network packet which is handled by the *transmit gadget* and measures the time from sending the packet until the response arrives. As described in Section 4, this round-trip time depends on the state of the microarchitectural element, and thus on the leaked bit.

As the network latency varies, the four steps have to be repeated to eliminate the noise caused by these fluctuations. Typically, the variance in latency follows a certain distribution depending on multiple factors, e.g., distance, number of hops, network congestion [26, 21, 13]. The number of repetitions depends mainly on the variance in network connection latency. Thus, depending on the latency distribution, the number of repetitions can be deduced using statistical methods. In Section 6.1, we evaluate this variant and provide empirically determined numbers for our attack setup.

```
if (x < array_length)
   access(array[x])
```

Listing 1.3: A NetSpectre gadget which can be used to break ASLR.

5.2 Remotely Breaking ASLR on the Target

If the attacker has no access to bit-leaking *NetSpectre gadgets*, it is possible to use a weaker *NetSpectre gadget* which does not leak the actual data but only information about the corresponding address. Such gadgets were not considered harmful for Spectre attacks, which already have local code execution, as ASLR does not protect against local attacks. However, in a remote scenario, it is very valuable to break ASLR. If such a *NetSpectre gadget* is found in a user-space program, it breaks ASLR for this process.

Listing 1.3 shows a *leak gadget* which we use to break ASLR in 3 steps:

1. Mistrain the branch predictor.
2. Out-of-bounds access to cache a known memory location.
3. Measure the execution time of a function via network to deduce whether the out-of-bounds access cached it.

The mistraining step is the same as for any Spectre attack, leading to speculative out-of-bounds accesses relative to the array. If the attacker provides an out-of-bounds value for `x` after mistraining, the array element indexed is speculatively accessed. Assuming a byte array and an (unsigned) 64-bit index, an attacker can (speculatively) access any memory location, as the index wraps around if the base address plus the index is larger than the virtual memory. If the byte at this memory location is valid and cacheable, the speculative execution will fetch the corresponding memory location into the cache. Thus, this gadget allows caching arbitrary memory locations which are valid in the current virtual memory, *i.e.*, every mapped memory location of the current application.

The attacker uses this gadget to cache a memory location at a known location, e.g., the vsyscall page which is mapped into every application at the same virtual address [15]. The attacker measures the execution time of a function accessing the now cached memory location. If it is faster, the out-of-bounds index actually cached an address used by this function. From the known address and the index value, *i.e.*, the relative offset to the known address, the address of the *leak gadget* can be calculated.

With an ASLR entropy of 30 b on Linux [42], there are 2^{30} possible offsets the attacker has to check. Due to the KPTI (formerly KAISER [23]) patches, no other page close to the vsyscall page is mapped in the user space. Consequently, in the 2^{30} possible offsets, there is only a single valid, and thus cacheable, offset. Hence, we can perform a binary search to find the correct offset, *i.e.*, speculatively try to load half of the possible offsets into the cache and check a single time. If the single valid offset was cached, the attacker chose the correct half. Otherwise, the attacker continues with the other half. This reduces the number of checks to defeat ASLR to only 30.

Although vsyscall is a legacy feature, we found it to be still enabled on Ubuntu 17.10 and Debian 9.4, the default operating system for VMs on the Google Cloud. Moreover, any other function or data can be used instead of vsyscall if the address is known. If the address of the *leak gadget* is known, it can be repeated to de-randomize any other function where its execution time of can be measured via the network. If the attacker knows a memory page at a fixed offset in the kernel, the same attack can be run on a *NetSpectre gadget* in the kernel to break KASLR.

6 Evaluation

In this section, we evaluate NetSpectre and the performance of our proof-of-concept implementation. Section 6.1 provides a qualitative evaluation and Section 6.2 a quantitative evaluation of our NetSpectre attacks. For the evaluation, we used laptops (Intel i5-4200M, i5-6200U, i7-8550U), as well as desktop PCs (Intel i7-6700K, i7-8700K), an unspecified Intel Xeon Skylake in the Google Cloud Platform, and an ARM A75.

6.1 Leakage

To evaluate NetSpectre on the different devices, we constructed a victim program which contains the same *leak gadget* and *transmit gadget* on all test platforms (cf. Section 3). We leaked known values from the victim to verify that our attack was successful and to determine how many measurements are necessary. Except for the cloud setup, all evaluations were done in a local lab environment. We used Spectre-PHT for all evaluations. However, other Spectre variants can be used in the same manner.

Desktop and Laptop Computers. Like other microarchitectural attacks, NetSpectre requires a large number of measurements to distinguish bits with a certain confidence (law of large numbers). On a local network, around 100 000 measurements are required to observe a difference clearly.

For our local attack, we had a gigabit connection between victim and attacker, a typical scenario in local networks and for network connections of dedicated and virtual servers. We measured a standard deviation of the network latency of 15.6 µs. Applying the three-sigma rule [54], in at least 88.8 % cases, the latency deviates ±46.8 µs from the average. This is nearly 3 orders of magnitude larger than the actual timing difference the attacker wants to measure, explaining the large number of measurements required.

Our proof-of-concept NetSpectre implementation leaks arbitrary bits from the victim by specifying an out-of-bounds index and comparing it with a user-provided value. Figure 7 shows the leakage of one byte using our proof-of-concept implementation. For every bit, we repeated the measurements 1 000 000 times. Although we only use a naïve threshold on the maximum of the histograms, we can clearly distinguish '0'-bits from '1'-bits (`array[x] <= y` and `array[x]`

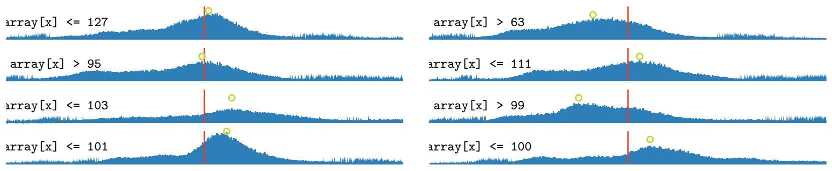

Fig. 7: Leaking the byte 100 (`01100100` in binary) bit by bit using a NetSpectre attack. The maximum of the histograms (green circle) can be separated using a simple threshold (red line). If the maximum is left of the threshold, the bit is interpreted as '1', otherwise as '0'.

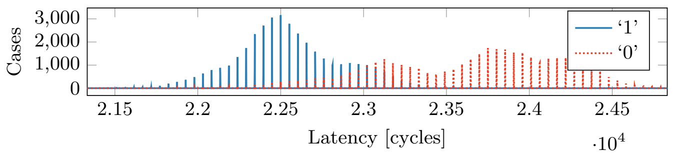

Fig. 8: Histogram of the measurements for a '0'-bit and a '1'-bit (`array[x] <= y` and `array[x] > y`) on an ARM Cortex A75. Although the times for both cases overlap, they are clearly distinguishable.

`> y`). More sophisticated methods, e.g., machine learning approaches, might be able to reduce the number of measurements further.

ARM Devices. Also in our evaluation on ARM devices, we used a wired network, as the network-latency varies too much in today's wireless connections. The ARM core we tested turned out to have a significantly higher variance in the network latency. We measured a standard deviation of the network latency of 128.5 µs. Again, with the three-sigma rule, we estimate that at least 88.8 % of the measurements are within ±385.5 µs.

Figure 8 shows two leaked bits, a '0'- and a '1'-bit (`array[x] <= y` and `array[x] > y`), of an ARM Cortex-A75 victim. Even with the higher variance in latency, thresholding allows separating the maxima of the histograms, *i.e.*, the attack works on ARM devices.

Cloud Instances. For the cloud instance, we tested our proof-of-concept implementation on the Google Cloud Platform. We created two VMs in the same region, one as the attacker, one as the victim. For both VMs, we used a default Ubuntu 16.04.4 LTS as the operating system. The measured standard deviation of the network latency was 52.3 µs. Thus, we estimate that at least 88.8 % of the measurements are in a range of ±156.9 µs.

To adapt for the higher variance in network latency, we increased the number of measurements to 20 000 000 per comparison. Figure 9 shows a (smoothed) histogram for both a '0'-bit and a '1'-bit (`array[x] <= y` and `array[x] > y`) on the Google Cloud VMs. Although there is still noise visible, it is possible to distinguish the two cases and thus perform a binary search to leak bit-by-bit of the value from the victim cloud VM.

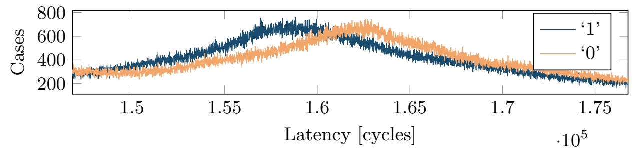

Fig. 9: Histogram of the measurements for the cases `array[x] <= y` and `array[x] > y` on two Google Cloud VMs with 20 000 000 measurements.

6.2 NetSpectre Performance

We evaluate the throughput and error rate of NetSpectre in this section.

Local Network. Attacks on the local network perform best, as the variance in network latency is significantly smaller than over the internet (cf. Section 6.1). In our setup, we repeat the measurement 1 000 000 times per bit to reliably leak bytes from the victim. On average, leaking one byte takes 30 min, which amounts to approximately 4 min per bit. Using the AVX covert channel instead of the cache reduces the required time to leak an entire byte to only 8 min. On average, we can break ASLR remotely within 2 h using the cache covert channel.

We used `stress -i 1 -d 1` for the experiments, to simulate a realistic environment. Although we expected our attack to work best on a completely idle server, we did not see any negative effects from the moderate server loads. In fact, they even slightly improved the performance. One reason for this is that a higher server load incurs a higher number of memory and cache accesses [1] and thus facilitates the cache thrashing (cf. Section 4), which is the performance bottleneck of our attack. Another reason is that a higher server load might exhaust execution ports required to calculate the bounds check in the leak gadget, thus increasing the chance that the CPU has to execute the condition speculatively.

Our NetSpectre attack in local networks is comparably slow. However, in particular, specialized malware attacks are often active over several months in local networks. Over such a time frame, the attacker can indeed leak all data of interest from a target system on the same network.

Cloud Network. We evaluated the performance in the Google cloud using two VMs. The two VMs have 2 virtual CPUs each, which enabled a 4 Gbit/s connection [20]. In this setup, we repeat the measurement 20 000 000 times per bit to get an error-free leakage of bytes. On average, leaking one byte takes 8 h for the cache covert channel, and 3 h for the AVX covert channel.

Despite the low performance, it shows that remote Spectre attacks are feasible between independent VMs in the public cloud. As specialized malware attacks often run for several weeks or months, such an extended time frame is sufficient to leak sensitive data, e.g., encryption keys or passwords.

Performance Improvements. For all measurements, we used commodity hardware in off-the-shelf laptops to measure the network-packet response time. Thus, there is additional latency (*i.e.*, noise) due to the latency of the operating system and network hardware of the attacker. Measuring the response time directly on the ethernet (or fiber) connection using dedicated hardware can drastically improve the attack performance. We expect that such a setup can easily reduce the time by a factor of 2 to 10.

7 Conclusion

In this paper, we presented NetSpectre, the first remote Spectre attack and the first Spectre attack which does not use a cache covert channel. With a remote Evict+Reload cache attack over network we can leak 15 bits per hour, with our new AVX-based covert channel even 60 bits per hour. We demonstrated NetSpectre on the Google cloud and in local networks, remotely leaking data and remotely breaking ASLR.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback and Anders Fogh, Halvar Flake, Jann Horn, Stefan Mangard, Jo Van Bulck, and Matt Miller for feedback on an early draft.

This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

1. Acıiçmez, O., Schindler, W., Koç, c.K.: Cache Based Remote Timing Attack on the AES. In: CT-RSA 2007 (2007)
2. Aly, H., ElGayyar, M.: Attacking aes using bernstein's attack on modern processors. In: International Conference on Cryptology in Africa (2013)
3. AMD, Inc.: Realview® Compilation Tools (2002)
4. AMD, Inc.: AMD64 Architecture Programmer's Manual (2017)
5. ARM Limited: CPU CORTEX-R8 (2009), https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r8
6. ARM Limited: ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. ARM Limited (2012)
7. ARM Limited: ARM Architecture Reference Manual ARMv8. ARM Limited (2013)
8. ARM Limited: Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism (2018)
9. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
10. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. arXiv:1903.01843 (2019)
11. Bulygin, Y.: Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly. ToorCon (2008)
12. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (2019)

13. Charneski, A.: Modeling network latency (2015), `https://blog.simiacryptus.com/2015/10/modeling-network-latency.html`
14. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. arXiv:1802.09085 (2018)
15. Corbet, J.: On vsyscalls and the vdso (2011), `https://lwn.net/Articles/446528/`
16. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: MICRO (2016)
17. Fog, A.: Test results for broadwell and skylake (2015), `http://www.agner.org/optimize/blog/read.php?i=415#415`
18. Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers (2016)
19. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. Journal of Cryptographic Engineering (2016)
20. Google: Egress throughput caps (2018), `https://cloud.google.com/compute/docs/networks-and-firewalls#egress_throughput_caps`
21. Goonatilake, R., Bachnak, R.A.: Modeling latency in a network distribution. Network and Communication Technologies **1**(2), 1 (2012)
22. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
23. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is Dead: Long Live KASLR. In: ESSoS (2017)
24. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA (2016)
25. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
26. Hopper, N., Vasserman, E.Y., Chan-Tin, E.: How much anonymity does network latency leak? TISSEC (2010)
27. Horn, J.: speculative execution, variant 4: speculative store bypass (2018)
28. Intel: Intel® 64 and IA-32 Architectures Software Developer′s Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z **253665** (2014)
29. Intel: Intel® 64 and IA-32 Architectures Software Developer′s Manual, Volume 1: Basic Architecture **253665** (2016)
30. Intel: Intel® 64 and IA-32 Architectures Software Developer′s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (325384) (2016)
31. Intel Newsroom: Advancing security at the silicon level (Mar 2018), `https://newsroom.intel.com/editorials/advancing-security-silicon-level/`
32. Intel Newsroom: Microcode revision guidance (Apr 2018), `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/04/microcode-update-guidance.pdf`
33. Jayasinghe, D., Fernando, J., Herath, R., Ragel, R.: Remote cache timing attack on advanced encryption standard and countermeasures. In: ICIAFs (2010)
34. Kiriansky, V., Waldspurger, C.: Speculative Buffer Overflows: Attacks and Defenses. arXiv:1807.03757 (2018)
35. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: S&P (2019)
36. Kocher, P.C.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)
37. Koruyeh, E.M., Khasawneh, K., Song, C., Abu-Ghazaleh, N.: Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT (2018)

38. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (2018)
39. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
40. Liu, W., Gao, D., Reiter, M.K.: On-demand time blurring to support side-channel defense. In: ESORICS (2017)
41. Maisuradze, G., Rossow, C.: ret2spec: Speculative execution using return stack buffers. In: CCS (2018)
42. Marco-Gisbert, H., Ripoll-Ripoll, I.: Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. BlackHat Asia (2016)
43. Masters, J.: Thoughts on NetSpectre (2018), `https://www.redhat.com/en/blog/thoughts-netspectre`
44. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: Cross-Cores Cache Covert Channel. In: DIMVA (2015)
45. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)
46. McCalpin, J.D.: Test results for Intel's Sandy Bridge processor (2015), `http://agner.org/optimize/blog/read.php?i=378#378`
47. Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Piessens, F., Sunar, B., Yarom, Y.: Fallout: Reading Kernel Writes From User Space. arXiv:1905.12701 (2019)
48. Oberman, S., Favor, G., Weber, F.: AMD 3DNow! technology: Architecture and implementations. IEEE Micro **19**(2), 37–48 (1999)
49. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
50. PaX Team: Address space layout randomization (ASLR) (2003)
51. Peleg, A., Weiser, U.: MMX technology extension to the Intel architecture. IEEE Micro **16**(4), 42–50 (1996)
52. Percival, C.: Cache missing for fun and profit. In: BSDCan (2005)
53. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
54. Pukelsheim, F.: The three sigma rule. The American Statistician (1994)
55. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue in-flight data load. In: S&P (2019)
56. Schwarz, M., Canella, C., Giner, L., Gruss, D.: Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. arXiv:1905.05725 (2019)
57. Schwarz, M., Gruss, D., Weiser, S., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks . In: DIMVA (2017)
58. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-privilege-boundary data sampling. arXiv:1905.05726 (2019)
59. Trippel, C., Lustig, D., Martonosi, M.: MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. arXiv:1802.03802 (2018)
60. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)
61. Zhao, X.j., Wang, T., Zheng, Y.: Cache Timing Attacks on Camellia Block Cipher. (2009)