NetSpectre: Read Arbitrary Memory over Network

Michael Schwarz
Graz University of Technology

Martin Schwarzl
Graz University of Technology

Moritz Lipp
Graz University of Technology

Daniel Gruss
Graz University of Technology

ABSTRACT

Speculative execution is a crucial cornerstone to the performance of modern processors. During speculative execution, the processor may perform operations the program usually would not perform. While the architectural effects and results of such operations are discarded if the speculative execution is aborted, microarchitectural side effects may remain. The recently published Spectre attacks exploit these side effects to read memory contents of other programs. However, Spectre attacks require some form of local code execution on the target system. Hence, systems where an attacker cannot run any code at all were, until now, thought to be safe.

In this paper, we present NetSpectre, a generic remote Spectre variant 1 attack. For this purpose, we demonstrate the first access-driven remote Evict+Reload cache attack over network, leaking 15 bits per hour. Beyond retrofitting existing attacks to a network scenario, we also demonstrate the first Spectre attack which does not use a cache covert channel. Instead, 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 verified that our NetSpectre attacks work in local-area networks as well as between virtual machines in the Google cloud.

NetSpectre marks a paradigm shift from local attacks, to remote attacks, exposing a much wider range and larger number of devices to Spectre attacks. Spectre attacks now must also be considered on devices which do not run any potentially attacker-controlled code at all. We show that especially in this remote scenario, attacks based on weaker gadgets which do not leak actual data, are still very powerful to break address-space layout randomization remotely. Several of the Spectre gadgets we discuss are more versatile than anticipated. In particular, value-thresholding is a technique we devise, which leaks a secret value without the typical bit selection mechanisms. We outline challenges for future research on Spectre attacks and Spectre mitigations.

Responsible Disclosure. **We disclosed our results to Intel on March 20th, 2018 and agreed on a disclosure date in late July 2018.**

1 INTRODUCTION

Modern computers are highly optimized for performance. However, these optimizations typically have side effects. Side-channel attacks observe these side effects and consequently deduce information which would usually not be accessible to the attacker. Software-based side-channel attacks are particularly unsettling since they do not require physical access to the device. Many of these attacks fall into the category of microarchitectural attacks, which exploit differences in the timing or the behavior, which are caused by microarchitectural elements.

Over the past 20 years, software-based microarchitectural attacks have evolved from theoretical attacks [49] on implementations of cryptographic algorithms [64], to more generic practical attacks [30, 79], and most recently to high potential threats [48, 55] breaking the fundamental memory and process isolation. Spectre [48] is a microarchitectural attack, tricking another program into speculatively executing an instruction sequence which leaves microarchitectural side effects. These side effects, in the case of all Spectre attacks demonstrated so far [15, 48, 58, 75], are timing differences caused by the pollution of data caches, *i.e.*, a traditional cache covert channel [56, 60].

Speculative execution, which is used in Spectre attacks, is a crucial cornerstone to the performance of modern processors. The branch prediction unit in modern processors makes an educated guess about which branch is taken and the processor then speculatively executes the expected instruction sequence following the predicted direction of the branch. By manipulating the branch prediction, Spectre tricks a target process into performing a sequence of memory accesses which leak secrets from chosen virtual memory locations to the attacker. This completely breaks confidentiality and renders virtually all security mechanisms on an affected system ineffective. Spectre variant 1 is the Spectre variant which affects the largest number of devices, mostly associated with misspeculation following bound checks. A code fragment performing first an operation such as a bound check and subsequently an operation with a microarchitectural side effect is called a “Spectre gadget”.

Spectre attacks have so far been demonstrated in JavaScript [48] and in native code [15, 48, 58, 75], 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 [15]. However, there are billions of 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, vendors are convinced that these systems are still safe and recommended to not take any action on these devices [42].

In this paper, we present NetSpectre, a new attack based on Spectre variant 1, 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 crafted requests to the victim and measures the response time to leak a secret value from the victim's memory.

We show that memory access latency, in general, can be 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. Based on this, we implemented the first access-driven remote cache attack, a remote variant of Evict+Reload called *Thrash+Reload*. Our remote *Thrash+Reload* attack is a significant leap forward from previous remote cache timing attacks on cryptographic algorithms [1, 5, 11, 16, 46, 82]. We facilitate this technique to retrofit existing Spectre attacks to our network-based scenario. This NetSpectre variant is able to leak 15 bits per hour from a vulnerable target system.

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

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, moving to a remote attack scenario, these weaker gadgets become 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 does not use the typical bit selection and memory reference mechanics as seen in previous Spectre attacks. Instead, value-thresholding exploits information leakage in comparisons directly, by using a divide-and-conquer approach similar to a binary search.

NetSpectre marks a paradigm shift from local attacks to remote attacks. This significantly broadens the range and increases the number of affected devices. In particular, Spectre attacks must also be considered a threat to the security of devices which do not run any untrusted attacker-controlled code. This shows that countermeasures must also be applied to these devices, which were previously thought to be safe. We propose a new alternative to Retpolines [77] which has a clearer structure. Future research on Spectre attacks and Spectre mitigations faces a series of challenges that we outline. These challenges indicate that the current defenses can only be temporary solutions since they only fix symptoms without addressing the root cause of the problem.

Contributions. The contributions of this work are:

(1) We present NetSpectre, a generic remote Spectre variant 1 attack. For this purpose, we demonstrate the first access-driven remote cache attack (Evict+Reload) over network, as a building block of NetSpectre.
(2) We demonstrate the first Spectre attack which does not utilize the cache. Instead, we present a new high-performance AVX-based covert channel which significantly improves the performance of remote Spectre attacks.
(3) We show that even weaker forms of local Spectre attacks, which are incapable of leaking actual data, are still very powerful in remote Spectre attacks enabling remote ASLR breaks without any code execution on the device.
(4) We show that Spectre gadgets can be more versatile than anticipated. Our technique *value-thresholding* allows obtaining a secret value without the typical bit selection and memory reference mechanics.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on speculative execution and microarchitectural attacks. In Section 3, we provide an overview of the full NetSpectre attack. In Section 4, we show how to build remote microarchitectural covert channels for use in NetSpectre attacks. In Section 5, we show how the building blocks are combined to extract memory contents over the network. In Section 6, we evaluate the performance of our attack. In Section 7, we discuss countermeasures against local and network-based Spectre attacks and outline challenges for future research. We conclude in Section 8.

2 BACKGROUND

In this section, we discuss out-of-order execution and a subset of out-of-order execution called speculative execution. We detail branch prediction, a building block of most speculative execution implementations. Finally, we will discuss known microarchitectural side-channel attacks as well as SIMD instructions as a better alternative for our use case.

2.1 Out-of-order and Speculative Execution

Modern processors do not strictly execute one instruction after another. Instead, modern processors have multiple execution units operating in parallel. The serial instruction stream is distributed over these execution units, leaving fewer processor resources unused. To retain the architecturally defined execution order, the processor has a so-called reorder buffer, which buffers operations until they are ready to be retired (made visible on the architectural level) in the order defined by the instruction stream. Hence, out-of-order execution lets the processor precompute the results and effects of instructions. Like simple pipeline processors, out-of-order

processors suffer from interrupts, since any precomputed results and effects have to be discarded. However, this is restricted to the architecturally visible state, and differences in the microarchitectural state may occur. Instructions which get precomputed but not retired are called transient instructions [48, 55].

Out-of-order execution on modern processors can typically run several hundred simple instructions ahead of the architecturally visible state. The actual number depends on the specific instructions and the size of the reorder buffer on the specific processor.

The instruction stream of almost every complex software is not purely linear but contains (conditional) branches. Consequently, the processor often does not know which direction of the branch to follow ahead of time, *i.e.*, which subsequent instructions to run out of order. In such cases, the processor uses prediction mechanisms to *speculatively* execute instructions along one of the paths. Hence, *speculative execution* is a strict subset of out-of-order execution. Correct predictions improve the performance and efficiency of the processor. Incorrect predictions require discarding any precomputed results and effects after the misprediction.

2.2 Branch Prediction

Branch prediction is the most common prediction mechanism causing speculative execution. Naturally, the performance and efficiency increase with the quality of the prediction. Therefore, modern processors incorporate a number of branch prediction mechanisms.

Intel [39] processors have prediction mechanisms for "Direct Calls and Jumps", "Indirect Calls and Jumps", and "Conditional Branches". These prediction mechanisms are implemented in different processor components, e.g., the Branch Target Buffer (BTB) [18, 53], the Branch History Buffer (BHB) [12], and the Return Stack Buffer (RSB) [20]. These buffers may be used in conjunction to obtain a good prediction [20]. Since branch-prediction logic is typically not shared across physical cores [22], the processor only learns from previous branches on the same core.

2.3 Microarchitectural Attacks

Most microarchitectural optimizations depend on the processed data or its location. As a consequence, observing the effect of an optimization (e.g., faster execution time) leaks information, e.g., about the data or its location.

Traditionally, microarchitectural attacks were split into two categories: side-channel attacks, which are non-destructive (passive), and fault attacks, which are destructive (active). Side-channel attacks are often used to build covert channels, *i.e.*, a communication between two colluding parties through a side channel.

Microarchitectural side-channel attacks were first explored for attacks on cryptographic algorithms [31, 49, 64, 68, 76]. More recently, generic practical attack techniques were developed and used against a wide range of attack targets, e.g., Flush+Reload [30, 31, 79].

Microarchitectural attacks are usually considered software-based attacks, as opposed to traditional side-channel attacks and fault attacks requiring physical access to the device. The most prominent example of a microarchitectural fault attack is Rowhammer, a hardware flaw in modern DRAM. Rowhammer enables modification of privileged DRAM memory locations by an unprivileged attacker.

Meltdown [55] and Spectre [48] are two recent microarchitectural attacks. They both use covert channels to transmit secrets, but the attacks themselves are no side-channel attacks. Since they are non-destructive, they appear to fall in neither of the two categories.

Meltdown [55] is a vulnerability present in many modern processors. It is the basis of a series of attacks, which all bypass the isolation provided by the `user_accessible` page-table bit (set to zero for kernel pages), e.g., different attacks on KASLR [28, 35, 45] that were discovered independently before Meltdown [55]. The full Meltdown attack allows reading arbitrary kernel memory [55].

Spectre attacks [48] exploit speculative execution, which is present in most modern processors. 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.

2.4 Cache Attacks

The largest group of microarchitectural attacks are cache attacks. Cache attacks exploit timing differences introduced by small memory buffers, called caches. These CPU caches hide memory access latencies by buffering frequently used data in small but fast in-processor memories. Modern CPUs have multiple cache levels that are either private per core or shared across cores.

Cache side-channel attacks were the first microarchitectural attacks. Different cache attack techniques have been proposed in the past, including Evict+Time [64], Prime+Probe [64, 68], and Flush+Reload [79]. Variations of these attacks are for instance Evict+Reload [30, 54], and Flush+Flush [29]. Flush+Reload attacks and its variants work on a cache-line granularity, as they rely on shared memory. Any cache line in shared memory will be a shared cache line in the inclusive last-level cache. In a Flush+Reload attack, the attacker constantly flushes a targeted memory location and measures the time it takes to reload the data. If the reload time is low, the attacker learns that another process has loaded the cache line into the cache. Various Flush+Reload attacks have been demonstrated, e.g., attacks on cryptographic algorithms [10, 44, 79], web server function calls [81], specific system activity [80], user input [30, 54, 73], and kernel addressing information [28]. Prime+Probe follows a similar principle, but only has a cache-set granularity. It works by occupying memory addresses and measuring when they are evicted from the cache. Hence, Prime+Probe attacks do not require any shared memory. Various Prime+Probe attacks have been demonstrated, e.g., attacks on cryptographic algorithms [43, 56, 64, 68], user input [54, 73], and kernel addressing information [35].

Cache timing side channels have also been demonstrated in remote timing attacks. Bernstein [11] proposed a remote timing attack against a weak implementation of the AES algorithm. The underlying timing difference was introduced from internal collisions in the algorithm and corresponding to that, a varying number of cache misses during AES computations. Subsequently, several works were published improving and reproducing this attack [1, 5, 46, 82].

A special use case of a side-channel attack is a covert channel. Here, the attacker controls both, the part that induces the side

effect, and the part that measures the side effect. This can be used to leak information from one security domain to another while bypassing any boundaries existing on the architectural level or above. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [29, 56, 61]. Meltdown [55] and Spectre [48] internally use a covert channel to transmit data from the transient execution to a persistent state.

2.5 SIMD Instructions

SIMD (single instruction multiple data) instructions allow performing an operation in parallel on multiple data values. SIMD instructions are available as instruction set extensions on a wide range of modern processors, e.g., the Intel MMX extension [36–38, 67], the AMD 3DNow! extension [3, 63], and the ARM VFP and NEON extensions [2, 7, 8]. On Intel, some of the SIMD instructions are processed by a dedicated SIMD unit within the processor core. However, to avoid wasting energy, the SIMD unit is turned off when it is not used. Consequently, to execute such SIMD instructions, the SIMD unit is first powered up, introducing a small latency on the first few SIMD instructions [20]. Liu [57] mentioned that some SIMD instructions can be used to improve bus-contention covert channels since they enable a more direct access the memory bus. However, so far, SIMD streaming instructions have not yet been used for pure SIMD covert channels or side-channel attacks.

2.6 Advanced Persistent Threats

The ever-increasing complexity in modern hardware and software is also reflected in modern malware. Especially targeted malware like Stuxnet [51], Duqu [9], or Flame [50], has proven to be extremely difficult to detect. Consequently, it can persist on a target system or network over periods of weeks or months. Hence, such malware is also known as "advanced persistent threats" (APTs) [74]. In this scenario, also slow covert channels that transmit a few bits to bytes per day, e.g., air-gap covert channels [32, 33], are highly practical, since they may run over a very long time. APTs are usually a combination of a set of concrete exploits, bypassing different security mechanisms to achieve an overall goal.

2.7 Address-Space Layout Randomization

One security mechanism present in modern operating systems is address-space layout randomization (ASLR) [66]. 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

In this section, we overview the NetSpectre attack based on a simple example. 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. We discuss *NetSpectre gadgets* in detail, 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 would not occur during strictly serialized in-order processing of the program's instructions, and which leak a victim's confidential information via a covert channel to an attacker. Spectre variant 1 induces speculative execution in the victim by mistraining a conditional branch, e.g., a bounds check. Spectre variant 2 induces speculative execution in the victim by maliciously injecting addresses into the branch-target buffer. Although our attack can utilize any Spectre variant, we focus on Spectre variant 1 as it is the most widespread. Moreover, according to Intel, in contrast to Meltdown and Spectre variant 2, variant 1 will not be fixed in hardware for the upcoming CPU generation [41].

Before the actual condition is known, the CPU predicts the most likely outcome of the condition 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 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 as if the processor would never have 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. The attacker must be able to send a large number of network packets to the victim. However, these do not necessarily have to be 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 processor, 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 valid and invalid values alternatingly to the exposed interface, e.g., valid and invalid network packets. 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 a bit stream at an attacker-controlled index, and changes some microarchitectural state depending on the state of the accessed bit. The *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, the attacker can observe this timing difference in the network packet response time. Spectre gadgets are commonly found in modern network drivers, network stacks, and network service implementation.

To illustrate the working principle of our NetSpectre attack, we consider a basic example similar to the original Spectre variant 1 example [48] in an adapted scenario: the code in Listing 1 is part of a function that is executed when a network packet is received. 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*.

```
if (x < bitstream_length)
   if(bitstream[x])
      flag = true
```

Listing 1: The conditional branch is part of a function executed when a network packet is processed.

The code fragment begins with a bound check on x, a best practice when developing secure software. In particular, this check prevents the processor from reading sensitive memory outside of bitstream. Otherwise, an out-of-bounds input x could trigger an exception or could cause the processor to access sensitive memory by supplying x = (address of a secret bit to read) − (base address of bitstream).

To exploit the microarchitectural state change during speculative execution in a remote attack, the attacker has to adapt the original Spectre attack. The attacker can remotely induce speculative execution as follows:

(1) The attacker sends multiple network packets such that the attacker-chosen value of x is always in bounds. This trains the branch predictor, increasing the chance that the branch predictor predicts the outcome of the comparison as true.
(2) The attacker sends a packet where x is out of bounds, such that bitstream[x] is a secret bit in the target's memory.
(3) Based on recent branch results of the condition, the branch predictor assumes the bounds check to be true, and the memory access is speculatively executed.

While changes in the architectural state are not committed after the correct result of the condition is resolved, changes in the microarchitectural state are not reverted. In the code in Listing 1 this means, that although the value of flag does not change, the cache state of flag does change. Only if the secret bit at bitstream[x] is set, flag is cached.

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 does leak a secret bit.

To complete the attack, the attacker measures the response time for every secret bit to leak. 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. 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 sufficient 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

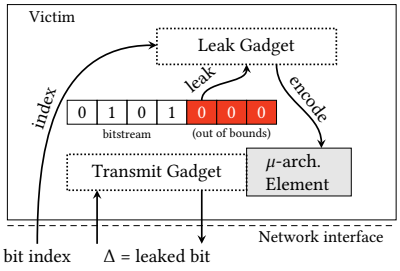

Figure 1: The interaction of the *NetSpectre gadget* types.

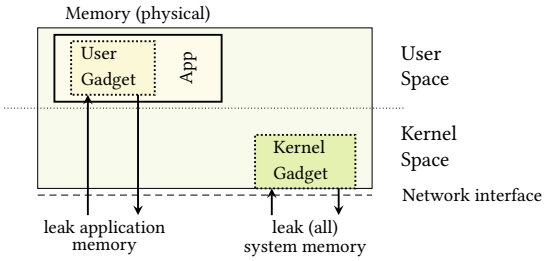

Figure 2: Depending on the gadget location, the attacker has access to either the memory of the entire corresponding application or the entire kernel memory, typically including the entire system memory.

interfaces, *i.e.*, both interfaces must be accessible for the attacker. Figure 1 illustrates the two types of gadgets, which will be described in more detail later in this section.

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: gadgets are either located in the user space or in the kernel space.

3.1.1 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 to reference a bit, 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 leveraging a *NetSpectre gadget* in the kernel can in general leak an arbitrary bit stored in memory.

3.1.2 Attacks on the User Space. Usually, network packets are not only handled by the kernel but are also 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 includes 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. But also, a remote server can 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 leveraging 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 secrets of multiple users. Especially for popular websites, this can easily affect thousands or millions of users.

3.2 Gadget type

We now discuss the different *NetSpectre gadgets*, namely 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.

3.2.1 Leak Gadget. The first type of gadget, the *leak gadget*, leaks secret data by changing a microarchitectural state depending on the value of a memory location that would not be accessible directly through any interface accessible to the attacker. Note that this state change happens on the victim device, and is not directly observable over the network.

A *leak gadget* gadget can leak a single bit, or even one or multiple bytes. 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). Kocher et al. [48] leaked the secret data with a byte-wise gadget. This simplifies the access to the secret data, as only byte indices have to be used, but complicates the recovery process, as 256 states have to be distinguished. With local Spectre attacks, the recovery process is implemented by the attacker, and thus a complex recovery process does not have any drawbacks but a slightly lower performance. The reason is that a larger number of side-channel tests (e.g., more Flush+Reload tests) have to be performed on the receiving side of the covert channel. Lipp et al. [55] showed that a transmission from out-of-order execution with single-bit covert channel can be significantly faster than a byte-wise or multi-byte covert channel in a similar attack. NetSpectre attacks have to rely on gadgets for the recovery process, slowing down the transmission significantly. A single-bit gadget does not only have several microarchitectural elements to choose from, but the data is also comparably easy to recover, and the data transmission is faster since fewer remote side-channel tests have to be performed for the covert channel transmission. Thus, we focus on single-bit *leak gadgets* in this paper. Single-bit *leak gadgets* can be as simple as shown in Listing 1. In this example, a value (`flag`) is cached if the bit at the attacker-chosen location is set. If the bit is not set, the cache state of the variable remains unchanged, *i.e.*, if it was previously uncached, it will not be cached. Hence, the attacker can use this gadget to leak secret bits into the microarchitectural state.

3.2.2 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.*, converting the microarchitectural state back into an architectural 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 a microarchitectural into an architectural 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. And indeed, the microarchitectural state will in some cases become visible to the attacker, 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 measurement of the microarchitectural state happens on a remote machine but exposes the microarchitectural 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 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

As described in the last section, a cornerstone of our NetSpectre attack is building a microarchitectural covert channel that exposes information to a remote attacker. Since in our scenario, the attacker cannot run any code on the target system, we assume the transmission happens through a *transmit gadget* whose execution can be triggered by the attacker. In this section, we demonstrate the first remote access-driven cache attack, *Thrash+Reload*, a variant of Evict+Reload. We show that with this remote cache attack, 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.

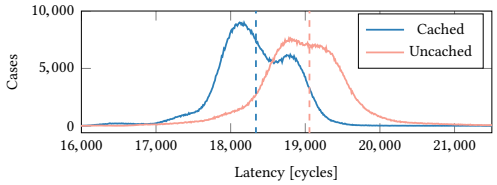

Figure 3: Measuring the response time of a simple *transmit gadget*, that accesses a certain variable. Only by performing a large number of measurements, the difference in the response timings depending on the cache state of the variable becomes visible. The average values of the two distributions are shown as dashed vertical lines.

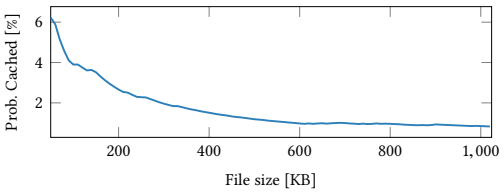

Figure 4: The probability that a specific variable is evicted from the victim's last-level cache by downloading a file from the victim (Intel i5-6200U). The larger the downloaded file, the higher the probability that the variable is evicted.

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. [48] leverage the cache as the microarchitectural element to encode the leaked data. This allows using well-known cache side-channel attacks, such as Flush+Reload [79] or Prime+Probe [64, 68] to deduce the microarchitectural state and thus the encoded data.

However, not only caches keep microarchitectural states which can be made visible on the architectural level. Methods to extract the microarchitectural state from elements such as DRAM [69], BTB [18], or RSB [13] are known. Generally, the receiver of every microarchitectural covert channel [22] can be used to transfer a microarchitectural state to an architectural state.

Mounting a Spectre attack by leveraging 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 [22, 29, 79]. If Flush+Reload is not applicable in a certain scenario, Prime+Probe is considered the next best choice [61, 72]. Consequently, all Spectre attacks published so far use either Flush+Reload [15, 48] or Prime+Probe [75].

To build our 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. Consequently, the response time will be influenced by the cache state of the variable used for the attack. The difference in the response time due to the cache state will be 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 influence of noise can be decreased by averaging over a large amount of network packets [1, 5, 11, 46, 82]. Hence, an attacker needs to average over a large number of measurements to obtain the secret value with acceptable confidence.

Figure 3 shows that the difference in the microarchitectural state is indeed visible when performing a large number of measurements. The average values 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 destroy 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 as Maurice et al. [60]. Hence, we call this technique *Thrash+Reload*. To thrash the entire cache without code execution, we again have to use an interface accessible via the network. In the simplest form, any packet sent from the victim to the attacker, e.g., a file download, has the chance to evict the 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 file with 590 kilobytes is already sufficient to evict the variable with a probability of $\geq 99\,\%$.

With a mechanism to distinguish cache hits and misses, as well as a mechanism to throw things out of 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 %.

In this paper, 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 [25, 30, 79].

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

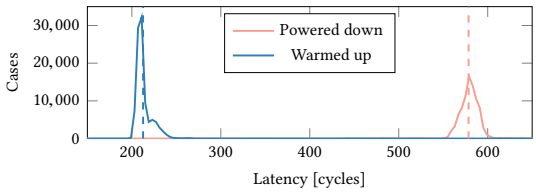

Figure 5: Differences in the execution time for AVX2 instructions (Intel i5-6200U). If the AVX2 unit is inactive (powered down), executing a 256-bit instruction takes on average 366 cycles longer than on an active AVX2 unit. The average values are shown as dashed vertical lines.

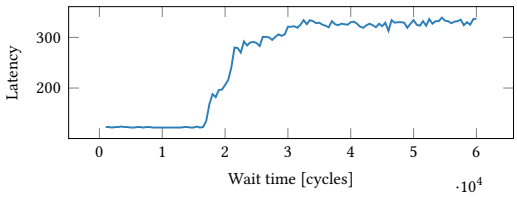

Figure 6: The number of cycles it takes to execute the `VPAND` instruction (including measurement overhead) after not using the AVX2 unit. After approximately 0.5 ms, the upper half of the AVX2 unit starts to power down, which increases the latency for subsequent AVX2 instructions. After approximately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

channel which allows transmitting information from speculative execution to an architectural state. Thus, we 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 [62]. If the unit is not used for more than 1 ms, it is powered down again [19].

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, resulting in a difference of 366 cycles. The difference 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 than the timing difference between cache hits and misses.

Similarly to the cache, reading the latency of an AVX2 instruction also destroys the encoded information. An attacker therefore requires a method to reset the AVX2 unit, *i.e.*, power down the upper half. In contrast to the cache, this is significantly 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 before the next measurement.

Figure 6 shows the execution time of a 256-bit 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, since the unit has to be powered up again and only emulates AVX2 in the meantime [19]. The AVX2 unit 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* leveraging AVX2 is similar to a *leak gadget* leveraging the cache. Listing 2 shows an example (pseudo-)code of an AVX2

```
if (x < bitstream_length)
   if(bitstream[x])
      _mm256_instruction();
```

Listing 2: An AVX2 NetSpectre gadget which encodes a bit using a 256-bit instruction.

leak gadget. The `_mm256_instruction` represents an arbitrary 256-bit AVX2 instruction, e.g., `_mm256_and_si256`. If the referenced bit `x` in the bit stream `bitstream` is set, the instruction is executed, and as a consequence, the upper half of the AVX2 unit is powered on. This is also true if the branch-prediction outcome was not correct and the AVX2 instruction is accessed during speculative execution. Note that there is no data dependency between the AVX2 instruction and either the bit stream or the index. Only the information whether an AVX2 instruction was executed is used to transmit the secret bit 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 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 was leaked, the function executes faster than if the upper half was not used before, *i.e.*, a '0'-bit was leaked.

With these building blocks, we can build an AVX-based covert channel. Our covert channel is 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, we can already see that it yields higher performance in our NetSpectre attack.

5 ATTACK VARIANTS

In this section, we describe two NetSpectre attack variants. The first attack allows extracting secret data bit-by-bit from the memory

of the target system. The second attacks allows defeating ASLR on the remote machine, paving the way for remote exploitation of bugs that ASLR would usually mitigate. We use gadgets based on Spectre variant 1 for illustrative purposes but this can naturally be done with any Spectre gadget that lies in a code path reached from handling of a remote packet.

5.1 Extracting Data from the Target System

With typical *NetSpectre gadgets* (cf. Section 3), the extraction process consists of 4 steps. Note that depending on the gadgets, the *leak gadget* and *transmit gadget* might be the same.

(1) Mistrain the branch predictor.
(2) Reset the state of the microarchitectural element.
(3) Leak a bit to the microarchitectural element.
(4) Expose state of the microarchitectural element to the network.

In step 1, the attacker mistrains the branch predictor of the victim to run a Spectre attack. To mistrain the branch predictor, the attacker leverages the *leak gadget* with valid indices. The valid indices ensure that the branch predictor learns to always take the branch, *i.e.*, the branch predictor speculates that the condition is true. Note that this step only relies on the *leak gadget*. There is no feedback to the attacker, and thus the microarchitectural state does not have to be reset or transmitted.

In step 2, the attacker has to reset the microarchitectural state to enable the encoding of leaked bits using a microarchitectural element. This step highly depends on the used microarchitectural element, e.g., when leveraging the cache, the attacker downloads a large file from the victim (cf. Figure 4), if AVX2 is used, the attacker simply waits for more than 1 ms. After this step, all requirements are satisfied to leak a bit from the victim.

In step 3, the attacker exploits the Spectre vulnerability 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 leverages 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 multiple times to eliminate the noise caused by these fluctuations. Typically, the variance in latency follows a certain distribution depending on multiple factors, such as distance, number of hops, network congestion [14, 24, 34]. The number of repetitions depends mainly on the variance in latency of the network connection. Thus, depending on the latency distribution, the number of repetitions can be deduced using statistical methods. In Section 6.1, we evaluate this attack variant and provide empirically determined numbers for our attack setup.

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

Listing 3: A NetSpectre gadget which can be leveraged to break ASLR.

5.2 Remotely Breaking ASLR on the Target System

If the attacker has no access to a 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 3 shows a simple *leak gadget* which is already sufficient to break ASLR. With the help of this gadget, breaking ASLR consists of 3 steps.

(1) Mistrain the branch predictor.
(2) Access an out-of-bounds index to cache a (known) memory location.
(3) Measure the execution time of a function via the network to deduce whether the out-of-bounds access cached a part of 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 at this index 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, it is cached after executing this gadget, as 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 [17]. The attacker then measures the execution time of a function accessing the now cached memory location, e.g., older versions of `time` or `gettimeofday`. If the function executes faster, the out-of-bounds array index actually cached a memory location used by this function. Thus, from the known address and the value of the array index, *i.e.*, the relative offset to the known address, the attacker can calculate the actual address of the *leak gadget*.

With an ASLR entropy of 30 b on Linux [59], there are 2^{30} possible offsets the attacker has to check. Due to the KPTI (formerly KAISER [27]) 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, and thus cacheable, 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 instances 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 also be repeated to de-randomize any other function if the execution time of this function can be measured via the network. If the attacker knows a memory page at a fixed offset in the kernel, the same attack can also 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 Core i5-4200M, i5-6200U, i7-8550U), as well as desktop PCs (Intel Core i7-6700K, i7-8700K), an unspecified Skylake-based Intel Xeon CPU in the Google Cloud Platform, and an ARM Cortex 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 variant 1 for all evaluations, however, other Spectre variants can be used in the same manner.

6.1.1 Desktop and Laptop Computers. In contrast to local Spectre attacks, where a single measurement can already be sufficient, NetSpectre attacks require a large number of measurements to distinguish bits with a certain confidence. Even on a local network, around 100 000 measurements are required to reduce the noise to a level where a difference between bits can be clearly seen. By repeating the attack, the noise is reduced, making it easier to distinguish the bits.

For our local attack we had a gigabit connection between the victim and the attacker, a typical scenario in local networks but also for network connections of dedicated servers and virtual servers. We measured a standard deviation of the network latency of 15.6 µs. Applying the three-sigma rule [70], 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 of a memory bitstream. 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. More sophisticated methods, e.g., machine learning approaches, might be able to further reduce the number of measurements.

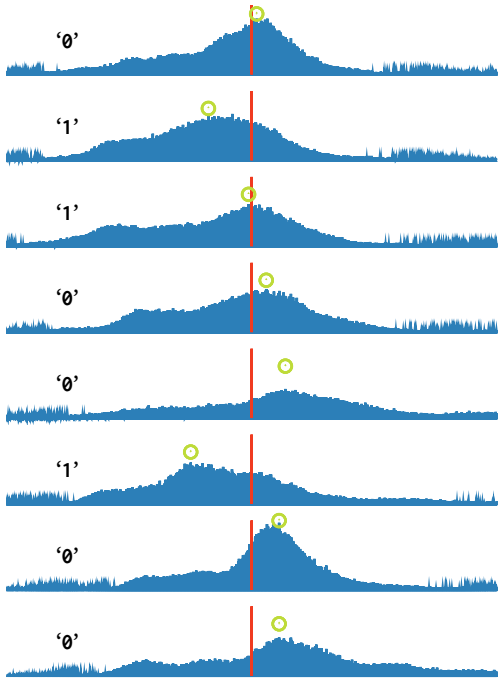

Figure 7: Leaking the byte 'd' (`01100100` in binary) bit by bit using a NetSpectre attack. The maximum of the histograms (green circle) can be separated using a simple threshold (red line). If the maximum is left of the threshold, the bit is interpreted as '1', otherwise it is interpreted as '0'.

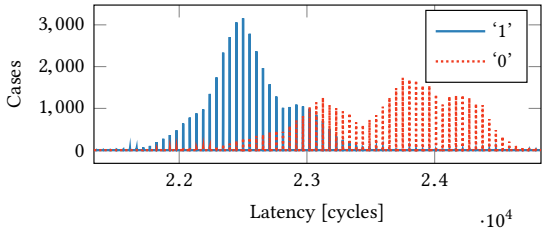

Figure 8: Histogram of the measurements for a '0'-bit and a '1'-bit on an ARM Cortex A75. Although the times for both cases overlap, they are clearly distinguishable.

6.1.2 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.

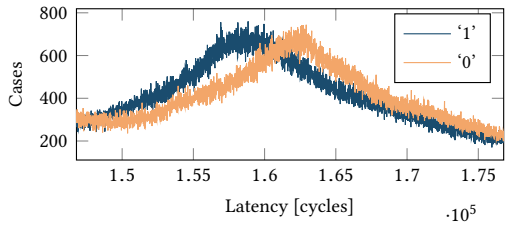

Figure 9: Histogram of the measurements for a '0'-bit and a '1'-bit on two Google Cloud virtual machines with 20 000 000 **measurements.**

Figure 8 shows two leaked bits—a '0'-bit and a '1'-bit—of an ARM Cortex A75 victim. Even with the higher variance in latency, simple thresholding allows separating the maxima of the histograms. Hence, the attack also works on ARM devices.

6.1.3 Cloud Instances. For the cloud instance, we tested our proof-of-concept implementation on the Google Cloud Platform. We created two virtual machine instances in the same region, one as the attacker, one as the victim. For both instances, 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. We verified that we can successfully leak data by running a NetSpectre attack between the two instances.

To adapt for the higher variance in network latency, we increased the number of measurements by a factor of 20, *i.e.*, every bit was measured 20 000 000 times. Figure 9 shows a (smoothed) histogram for both a '0'-bit and a '1'-bit on the Google Cloud instances. Although there is still noise visible, it is possible to distinguish the bits and thus leak arbitrary bits from the victim cloud instance.

6.2 NetSpectre Performance

To evaluate the performance of NetSpectre, we leaked known values from a target device. This allows us to not only determine how fast an attacker can leak memory, but also to determine the bit-error rate, *i.e.*, how many bit errors to expect.

6.2.1 Local Network. Attacks on the local network achieve the best performance, as the variance in network latency is significantly smaller than the variance over the internet (cf. Section 6.1.3). In our lab setup, we repeat the measurement 1 000 000 times per bit to be able 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.

To break ASLR, we require the cache covert channel. On average, this allows breaking the randomization remotely within 2 h.

We used `stress -i 1 -d 1` for the experiments, to simulate a realistic environment. Although we would have 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 attack 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 bottle neck 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, e.g., APTs, 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.

6.2.2 Cloud Network. We evaluated the performance in the cloud using two virtual machines instances on the Google Cloud. These virtual machines have a fast network connection. We configured the two instances to each use 2 virtual CPUs, which enabled a 4 Gbit/s connection [23]. 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.

While this is comparably slow, it shows that remote Spectre attacks are feasible between independent instances in the public cloud. In particular, APTs typically run for several weeks or months. Such an extended time frame is clearly sufficient to leak sensitive data, such as encryption keys or passwords, using the NetSpectre attack in a cloud environment.

7 CHALLENGES OF MITIGATING SPECTRE

In this section, we discuss limitations of state-of-the-art countermeasures against Spectre, and how they do not fully prevent NetSpectre attacks. Furthermore, we discuss how NetSpectre attacks can be prevented on the network layer. Finally, we outline challenges for future research on Spectre attacks as well as Spectre mitigations.

7.1 State-of-the-art Spectre Countermeasures

Due to the different origins, Spectre variant 1 and variant 2 are mitigated using separate countermeasures. Intel released microcode updates to prevent the cross-process and cross-privilege mistraining of indirect branches typical for Spectre variant 2 attacks. There are no microcode updates to prevent mistraining of direct branches, since this is easy to do in-place, *i.e.*, in the same privilege level and the same process context. For Spectre variant 1 attacks, a series of pure software countermeasures have been proposed.

Intel and AMD recommend using the `lfence` instruction as a speculation barrier [4, 40]. This instruction has to be inserted after security-critical bounds check to stop the speculative execution. However, adding this to every bounds check has a significant performance overhead [40].

Moreover, our experiments showed that `lfences` do stop the speculative execution but not speculative code fetches and other microarchitectural behaviors that occur pre-execution, such as powering up of the AVX functional units, instruction cache fills, and TLB fills. According to our experiments, `lfences` do work against traditional Spectre gadgets but not against all Spectre gadgets we use in this paper, cf. Listing 2 and Listing 3, which can already leak information through microarchitectural behaviors that occur pre-execution. However, we believe there are ways to use `lfences` in a way that mitigates the leakage.

Microsoft implements an automatic detection of vulnerable code paths, *i.e.*, Spectre gadgets, in its compiler to limit the speculation barrier to these gadgets [65]. However, Kocher [47] showed that the automated analysis misses many gadgets. As Microsoft only uses a blacklist for known gadgets [47], many gadgets, in particular gadgets which are not typical (e.g., gadgets to break ASLR), are not automatically safeguarded by the compiler.

In the Linux kernel, exploitable gadgets are identified manually and with the help of static code analyzers [52]. Similarly to the compiler-based approach, this requires a complete understanding of which code snippets are exploitable.

Finally, until now it was widely overlooked that indirect branch mistraining (Spectre variant 2) is also possible in-place. However, the attack possibilities are much more constrained with in-place mistraining.

7.2 Network-layer Countermeasures

As NetSpectre is a network-based attack, it cannot only be prevented by mitigating Spectre but also through countermeasures on the network layer. A trivial NetSpectre attack can easily be detected by a DDoS protection, as multiple thousand identical packets are sent from the same source. However, an attacker can choose any trade-off between packets per second and leaked bits per second. Thus, the speed at which bits are leaked can simply be reduced below the threshold that the DDoS monitoring can detect. This is true for any monitoring which tries to detect ongoing attacks, e.g., intrusion detection systems. Although the attack is theoretically not prevented, at some point the attack becomes infeasible, as the time required to leak a bit increases drastically.

Another method to mitigate NetSpectre is to add artificial noise to the network latency. As the number of measurements depends on the variance in network latency, additional noise requires an attacker to perform more measurements. Thus, if the variance in network latency is high enough, NetSpectre attacks become infeasible due to the large number of measurements required.

Both approaches may mitigate NetSpectre attacks in practice. However, as attackers can adapt and improve attacks, it is not safe to assume that noise levels and monitoring thresholds chosen now will still be valid in the near future.

7.3 Future Research Challenges

As discussed in the previous sections, Spectre is far from being a solved case. The currently proposed mitigations merely fix symptoms without directly addressing the root cause, the imbalanced trade-off between performance and security that led to the speculative execution we currently have. We identified 5 challenges (C1 to C5) for future work on Spectre attacks and mitigations.

***C1**: Gadgets are more versatile than anticipated.* In particular the gadgets we use to break ASLR have not been considered dangerous so far. Also the AVX-based gadgets we use were not considered so far. Gadgets may also consist of many small code pieces that pass on secret values until at a later point the secret value is leaked to the attacker. Since the building block of Spectre that exfiltrates the information to the attacker is a covert channel, it appears the underlying problem of identifying all gadgets may be reduced to the problem of identifying all covert channels. Currently, we have no technique to identify all covert channels in a system.

***C2**: Automatically safeguarding all gadgets is not trivial.* For Spectre variant 1 the proposed solution is to use speculation barriers [4, 40]. As we cannot expect every developer to identify vulnerable gadgets and correctly fix them, state-of-the-art solutions try to automatically detect vulnerable gadgets and fix them at compile time [65]. At the moment it is not clear whether static code analysis is sufficient to detect all vulnerable gadgets, especially if they are scattered across functions. In such complex scenarios, dynamic analysis might lead to better results. However, dynamic analysis naturally suffers from incompleteness, as certain parts of the program may not be reached in the dynamic analysis. Furthermore, it might be possible that the compiler produces Spectre gadgets which are not visible in the source code, as it can happen with e.g., double-fetch bugs [71]. This would hardly be detected upfront and completely undermine the security measures taken.

***C3**: Blacklisting is inherently incomplete.* Current approaches rely on blacklists to automatically patch exploitable gadgets [65]. However, this implies that we understand exactly which code fragments are exploitable and which are not. As this paper shows, gadgets can look different than anticipated, showing the incompleteness of the blacklist approach. Inverting the logic might be a better direction, *i.e.*, using whitelists of (provably) unexploitable gadgets instead of blacklists. However, this would require a substantial amount of research on proving non-exploitability of code fragments.

***C4**: Cross-process and cross-privilege-level mistraining is easier to solve than in-place mistraining.* Current countermeasures mainly aim at preventing Spectre attacks across process boundaries and there especially across privilege levels [6, 40, 41]. However, as shown in this paper, such countermeasures are ineffective if the mistraining happens in-place inside the same process. This method is not only applicable to Spectre variant 1, but also to Spectre variant 2 [26]. Retpoline [77] is currently the only mitigation that protects against these Spectre variant 2 attacks by effectively stopping any further speculation by the processor. However, Retpoline is not a perfect solution, as it incurs significant performance overheads and adds another side channel [21].

If an attacker can only poison branches with valid branch targets inside the same process, *i.e.*, all microcode updates applied, Retpoline can be replaced by a more simple construct we propose. We propose to insert speculation barriers at every possible call target. This is a much clearer structure than with Retpolines. Thus, every misspeculated indirect call immediately aborts before actually executing code. For direct calls, the compiler can jump just beyond the speculation barrier to have less performance impact. Still, the overall performance impact of this solution, just like Retpoline, would be significant. It remains unclear whether Spectre attacks within the same process can be fully prevented without high performance overheads and without introducing new problems.

***C5**: Security mechanisms may have unwanted side effects.* The Retpoline patch basically hides the target of indirect calls from the CPU by fiddling with the return values on the stack. However, this leads to side effects with other security mechanisms, as a Retpoline

behaves similarly to an exploit changing the control flow. Especially security mechanisms such as control-flow integrity have to be adapted [78] to not falsely detect Retpolines as attacks. Still, the question arises how Spectre mitigations interact with other CFI implementations, especially in hardware, as well as other security mechanisms and whether we have to accept trade-offs when combining security mechanisms. In general, we need to investigate which security mechanisms could have detrimental side effects that outweigh the gains in security.

8 CONCLUSION

In this paper, we presented NetSpectre, the first remote Spectre variant 1 attack. We demonstrated the first access-driven remote Evict+Reload cache attack over network, with a performance of 15 bits per hour. We also demonstrated the first Spectre attack which does not use a cache covert channel. In particular, in a remote Spectre attack, our novel high-performance AVX-based covert channel performs significantly better than the remote cache covert channel. Our NetSpectre attack in combination with the AVX-based covert channel leaks 60 bits per hour from the target system. We verified NetSpectre in local networks as well as in the Google cloud.

NetSpectre marks a paradigm shift for Spectre attacks, from local attacks to remote attacks. With our NetSpectre attacks, a much wider range and larger number of devices are exposed to Spectre attacks. Spectre attacks now must also be considered on devices which do not run any potentially attacker-controlled code at all. We demonstrate that in a remote attack, NetSpectre can be used to defeat address-space layout randomization on the remote system. As we discussed in this paper, there are a series of open challenges for future research on Spectre attacks and Spectre mitigations.

ACKNOWLEDGMENTS

We would like to thank Anders Fogh, Halvar Flake, Jann Horn, Stefan Mangard, 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] Onur Acıiçmez, Werner Schindler, and Çetin Kaya Koç. 2007. Cache Based Remote Timing Attack on the AES. In *CT-RSA 2007*.
[2] Advanced Micro Devices, Inc. 2002. RealView® Compilation Tools. (2002).
[3] Advanced Micro Devices, Inc. 2017. AMD64 Architecture Programmer's Manual. (2017).
[4] Advanced Micro Devices, Inc. 2018. Software Techniques for Managing Speculation on AMD Processors. http://developer.amd.com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors.pdf. (2018).
[5] Hassan Aly and Mohammed ElGayyar. 2013. Attacking aes using bernstein's attack on modern processors. In *International Conference on Cryptology in Africa*.
[6] ARM. 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/security-update. (2018).
[7] ARM Limited. 2012. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition.* ARM Limited.
[8] ARM Limited. 2013. *ARM Architecture Reference Manual ARMv8.* ARM Limited.
[9] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. 2012. Duqu: Analysis, detection, and lessons learned. In *ACM European Workshop on System Security (EuroSec)*, Vol. 2012.
[10] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES'14*.
[11] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[12] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2017. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. Cryptology ePrint Archive, Report 2017/968. (2017).
[13] Yuriy Bulygin. 2008. Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly. *Proceedings of the ToorCon* (2008).
[14] Andrew Charneski. 2015. Modeling Network Latency. (2015). https://blog.simiacryptus.com/2015/10/modeling-network-latency.html
[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv:1802.09085* (2018).
[16] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The last mile: An empirical study of timing channels on seL4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.
[17] Jonathan Corbet. 2011. On vsyscalls and the vDSO. (2011). https://lwn.net/Articles/446528/
[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *International Symposium on Microarchitecture (MICRO)*.
[19] Agner Fog. 2015. Test results for Broadwell and Skylake. (2015). http://www.agner.org/optimize/blog/read.php?i=415#415
[20] Agner Fog. 2016. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* http://www.agner.org/optimize/microarchitecture.pdf
[21] Anders Fogh. 2018. In debt to Retpoline. (2018). https://cyber.wtf/2018/02/13/in-debt-to-retpoline/
[22] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).
[23] Google. 2018. Egress throughput caps. (2018). https://cloud.google.com/compute/docs/networks-and-firewalls#egress_throughput_caps
[24] Rohitha Goonatilake and Rafic A Bachnak. 2012. Modeling latency in a network distribution. *Network and Communication Technologies* 1, 2 (2012), 1.
[25] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
[26] Daniel Gruss, Moritz Lipp, and Michael Schwarz. 2018. Beyond Belief: The Case of Spectre and Meltdown. *Bluehat IL* (Jan. 2018). http://www.bluehatil.com/files/Beyond%20Belief%20-%20The%20Case%20of%20Spectre%20and%20Meltdown.pdf
[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.
[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.
[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.
[31] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*.
[32] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. 2014. AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In *9th International Conference on Malicious and Unwanted: Software The Americas (MALWARE)*.
[33] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. 2015. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In *IEEE 28th Computer Security Foundations Symposium (CSF)*.
[34] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. 2010. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)* (2010).
[35] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.
[36] Intel. 2014. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 253665 (2014).
[37] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 253665 (2016).
[38] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 253665 (2016).
[39] Intel. 2017. Intel 64 and IA-32 Architectures Optimization Reference Manual. (2017).
[40] Intel Corp. 2018. Intel Analysis of Speculative Execution Side Channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf. (Jan. 2018).
[41] Intel Newsroom. 2018. Advancing Security at the Silicon Level. (March 2018). https://newsroom.intel.com/editorials/advancing-security-silicon-level/

[42] Intel Newsroom. 2018. Microcode Revision Guidance. (April 2018). https://newsroom.intel.com/wp-content/uploads/sites/11/2018/04/microcode-update-guidance.pdf

[43] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P'15*.

[44] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14*.

[45] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*.

[46] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. 2010. Remote cache timing attack on advanced encryption standard and countermeasures. In *5th International Conference on Information and Automation for Sustainability (ICIAFs)*.

[47] Paul Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler. (2018). https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[48] 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:1801.01203* (2018).

[49] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

[50] David Kushner. 2013. The real story of stuxnet. *IEEE Spectrum* 50, 3 (2013), 48–53.

[51] Ralph Langner. 2011. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy* 9, 3 (2011), 49–51.

[52] Julia Lawall. 2018. Re: [RFC PATCH] asm/generic: introduce if_nospec and nospec_barrier. (2018). https://lwn.net/Articles/743287/

[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.

[54] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[56] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy – SP*. IEEE Computer Society, 605–622.

[57] Weijie Liu, Debin Gao, and Michael K Reiter. 2017. On-demand time blurring to support side-channel defense. In *ESORICS*.

[58] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv:1801.04084* (2018).

[59] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. 2016. Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. *BlackHat Asia* (2016).

[60] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.

[61] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[62] John D. McCalpin. 2015. Test results for Intel's Sandy Bridge processor. (2015). http://agner.org/optimize/blog/read.php?i=378#378

[63] Stuart Oberman, Greg Favor, and Fred Weber. 1999. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* 19, 2 (1999), 37–48.

[64] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[65] Andrew Pardoe. 2018. Spectre mitigations in MSVC. (2018). https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[66] PaX Team. 2003. Address space layout randomization (ASLR). (2003). http://pax.grsecurity.net/docs/aslr.txt

[67] Alex Peleg and Uri Weiser. 1996. MMX technology extension to the Intel architecture. *IEEE Micro* 16, 4 (1996), 42–50.

[68] Colin Percival. 2005. Cache missing for fun and profit. In *Proceedings of BSDCan*.

[69] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.

[70] Friedrich Pukelsheim. 1994. The three sigma rule. *The American Statistician* (1994).

[71] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS* (2018).

[72] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks . In *DIMVA'17*.

[73] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[74] Colin Tankard. 2011. Advanced persistent threats and how to monitor and deter them. *Network security* 2011, 8 (2011), 16–19.

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

[76] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES'03*. 62–76.

[77] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. (2018).

[78] Florian Weimer. 2018. Retpolines and CFI. (2018). https://gcc.gnu.org/ml/gcc/2018-01/msg00160.html

[79] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[80] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 858–870.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS'14*.

[82] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. 2009. Cache Timing Attacks on Camellia Block Cipher. Cryptology ePrint Archive, Report 2009/354. (2009).