Martin Schwarzl

NetSpectre - Leaking Arbitrary Memory from Remote Systems

Master's Thesis

to achieve the university degree of

Dipl. Ing.

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Daniel Gruß

Institute for Applied Information Processing and Communications

Advisor
Michael Schwarz

Faculty of Computer Science and Biomedical Engineering

Graz, March 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum	Unterschrift

Abstract

In January 2018, a CPU vulnerability called Spectre was presented, which exploits side-channel information of speculative execution in CPUs to read arbitrary virtual memory from programs. Speculative execution is needed to meet the currently high demands on CPU performance. Instructions are executed speculatively to enable faster execution times. During speculative execution, side effects occur, for example, in the CPU cache, which can be measured and used to read any memory. Over one billion smartphones, desktop and notebook CPUs are affected by this bug. Since the bug is located directly in the CPU, the hardware would have to be replaced. But exchanging the hardware also means a new design of the speculative execution.

In this master thesis, we show a novel attack which exploits Spectre variant 1 over the network. This attack makes it possible to read sensitive data from the memory of target systems without system access and thus possible code execution. Sensitive data is, for instance, secret user credentials, credit card data or encryption keys. Two types of code snippets, so-called gadgets, are required to carry out the attack. The first gadget is needed to mistrain the speculative execution, so that a certain conditional branch is always taken. The speculative execution then accesses data whose access is not permitted. The second gadget allows an attacker to access the same data address. Since the data is already in the CPU cache from the first access, the second access is faster than the first. This timing difference can be measured and recognized over the network. Using this information we have developed the first variant of NetSpectre, which is the cache-based attack. Here, we adapt the well-known cache attack Evict+Reload and make it network-compatible. This variant makes it possible to read up to 15 bits per hour. Furthermore, we present the first Advanced Vector Instructions based remote covert channel. With this approach, we are able to leak up to 60 bits per hour. In addition, we present a technique with which we can break Address Space Layout Randomization (ASLR) via NetSpectre gadgets. This allows memory randomization to be bypassed. We verified this attack in local networks, virtual machines and the Google Cloud Platform. We also discuss current countermeasures for Spectre and NetSpectre. The content of this thesis will be presented as a talk at Black Hat Asia 2019.

Keywords: operating systems, speculative execution, branch prediction

Kurzfassung

Im Jänner 2018 wurde eine CPU-Schwachstelle namens Spectre präsentiert, welche Seitenkanalinformation der spekulativen Ausführung in CPUs ausnützt, um beliebigen virtuellen Speicher von Programmen auszulesen. Spekulative Ausführung wird benötigt, um die derzeit enorm hohen Anforderungen an CPU Performance zu gewährleisten. Instruktionen werden hierbei im Vorhinein spekulativ ausgeführt, um schnellere Ausführungszeiten zu ermöglichen. Während der spekulativen Ausführung entstehen Seiteneffekte zum Beispiel im CPU-Cache welche gemessen werden können und genutzt um beliebigen Speicher auszulesen. Über eine Milliarde Smartphones, Desktop und Notebook CPUs sind von dieser Lücke betroffen. Da der Bug sich direkt in der CPU befindet, müsste die Hardware getauscht werden. Tausch der Hardware bedeutet aber auch ein neues Design der spekulativen Ausführung. In dieser Masterarbeit zeigen wir eine neue Variante, um Spectre Variante 1 über das Netzwerk auszunützen. Diese Attacke ermöglicht es, ohne Systemzugriff und damit möglicher Code Execution, sensitive Daten aus dem Speicher von Zielsystemen auszulesen. Sensitive Daten sind beispielsweise geheime Nutzerdaten und Verschlüsselungsschlüssel. Um die Attacke durchzuführen, werden zwei Arten von Codeteilen, sogenannte Gadgets benötigt. Das erste Gadget wird benötigt um die spekulative Ausführung zu trainieren, sodass diese einen gewissen bedingten Sprung im Programm immer spekulativ ausführt. Durch die spekulative Ausführung wird dann auf Daten zugegriffen, deren Zugriff nicht erlaubt ist. Mit dem zweiten Gadget wird nochmals auf dieselbe Datenadresse zugegriffen. Da die Daten schon vom ersten Zugriff im CPU-Cache liegen, ist der zweite Zugriff schneller als der Erste. Diesen Zeitunterschied kann man messen und über das Netzwerk erkennen. Mit dieser Information haben wir die erste Variante von NetSpectre entwickelt, die den CPU-Cache attackiert. Hierbei adaptieren wir die bekannte Cache-Attacke Evict+Reload und machen sie netzwerkfähig. Diese Variante ermöglicht es, bis zu 15 Bits pro Stunde auszulesen. Des Weiteren präsentieren wir den ersten Advanced Vector Instructions basierten remote Covert Channel. Mit diesem sind wir in der Lage bis zu 60 bits in der Stunde auszulesen. Außerdem präsentieren wir eine Technik, mit der wir Address-Space-Layout-Randomization brechen können über NetSpectre Gadgets. Das erlaubt es Speicherrandomisierung zu umgehen und im Falle einer weiteren Schwachstelle im Code diese auszunützen. Der Inhalt dieser Masterarbeit wird bei der Black Hat Asia 2019 als Talk präsentiert. Die Ergebnisse dieser Attacke wurden in lokalen Netzwerken, virtuellen Maschinen und in der Google-Cloud Platform verifiziert. Des Weiteren diskutieren wir aktuelle Gegenmaßnahmen für Spectre und NetSpectre.

Stichwörter: Betriebssysteme, Spekulative Ausführung, Branch Prediction

Acknowledgements

First of all I want to thank my advisor Michael Schwarz for his constant support. Especially for the helpful discussions, tips and the reviews of this thesis.

I also want to thank Daniel Gruss and Moritz Lipp for meaningful discussions and their support.

Finally, I also want to especially thank my family for their way of keeping me motivated during my studies.

Martin Schwarzl

Contents

Chapter 1

Introduction

Side-channel attacks received more and more popularity during the last couple of years. These attacks use information like timings, power consumptions of a device, faulty implementations or even electromagnetic radiations [85] to leak sensitive data. At first, those attacks were quite theoretical and only got slow proof-of-concepts implementations. Nowadays, fast and stealthy attacks are practical and can be used to read arbitrary system memory [8, 49, 54]. Side-channel attacks are hard to detect and even harder to mitigate [26, 53, 69].

Cache side-channel attacks exploit a timing difference of about 150 CPU cycles which arises when accessing cached memory and uncached memory. Cache side-channel attacks are in the meantime a powerful weapon to attack software. They have been used to spy keystrokes on libraries [28] for instance keystrokes on SSH [81], attacking cryptography [1, 37, 40, 50, 66, 67, 91], building of stealthy and cross-VM covert channels [27, 38, 46, 69, 89]. These cross VM side-channels, also work on most of the common cloud providers [89]. There is a lot of research going on to detect and mitigate cache side-channel attacks [22]. For instance, cryptographic primitives like AES already got constant time implementations to mitigate cache attacks [37, 40, 44, 66].

Speculative execution is one of the essential components in modern CPUs to meet the high-performance requirements. In combination with branch prediction, it enables faster execution times. A branch predictor guesses whether a particular branch is taken or not taken. The predictions are then speculatively executed and committed to the architecture if the prediction was correct. If the prediction was incorrect, the results get reverted and are not applied to the architecture. However, this speculative execution might perform *transient instructions*, which change the microarchitectural state. The changes in the microarchitectural can be observed in form of the timing for instance in the cache [49].

In 2018, four *transient execution* attacks, namely Meltdown, Spectre, Foreshadow and Foreshadow NG [49, 54, 83, 88], got published. All attacks allow an attacker to leak sensitive information, which is supposed to be non-accessible. Affected are desktop PCs, notebooks, servers and smartphones CPUs from the common manufacturers Intel, AMD, ARM [8]. We can assume that billions of devices are still affected by these issues since the problem lies in the hardware [49, 54]. Spectre exploits the prediction components from the branch prediction in combination with speculative execution. The first Spectre variant known as Spectre V1 exploits the Pattern History Table [49]. The other versions exploit the Branch Target Buffer (Spectre V2), Store-To-Load-Forward and Return Stack Buffer [8]. Furthermore, the attack variants were validated on Intel SGX [10]. In contrast to Spectre, Meltdown exploits transiently executed out-of-order instructions [54]. Those instructions perform an illegal memory access and afterwards raise an exception [54]. The Foreshadow attacks exploit the L1 cache [83, 88] to bypass hypervisors and Intel's SGX.

Multiple countermeasures for Spectre were already proposed in the form of microcode updates from Intel [36]. These updates fix the possibility of leaking kernelspace memory from userspace applications via Spectre. These patches weaken the attacking potential of Spectre. However, it is still possible to exploit userspace applications [36]. To leak kernelspace memory, for instance, a driver with vulnerable code is needed.

The recommended way from Intel, ARM and AMD to mitigate Spectre is to use memory barriers, which disallow speculative execution for a specific code section [36,86]. However, these memory barriers need to be applied for each critical code section where sensitive data is accessed. Another possibility is to do a constant cache flushing after each critical sections [49]. However, this has an impact on the execution speed of a program [49]. SafeSpec [45] and InvisiSpec [90] were proposed as a hardware solution to migitate side-channel attacks in general. The worst impact on performance is a total deactivation of caches and out-of-order execution. Nevertheless, deactivation mitigates these types of side-channel attacks. At the moment, there is no satisfying solution which fits both performance and also security requirements.

This thesis focuses on exploiting Spectre over a networking aspect, which is a new attack vector. We call this new attack vector **NetSpectre** [77]. We define code snippets (gadgets) which need to exist and have to be used in the server program. In the attack, we first evaluate the cache timing difference for a client-server connection. As data encoding, we choose a binary format. Based on this, we build network histograms which give us a threshold to distinguish between a zero and one bit. Then, we build a proof-of-concept implementation where the client has access to a public resource from the server. Using the NetSpectre attack, the client is capable of leaking private information from the server. Furthermore, we discovered that Advanced Vector Instructions (AVX) instructions also leak timing information. AVX instructions behave differently in timing when the unit is warmed up or cold. Using this side channel, we can port the attack from a cache-based approach to an AVX-based approach. We build the first AVX-based covert channel which is more performant than the cache-based approach. With this approach, we are capable of leaking **60** bits per hour in a local network environment. Then, we demonstrate a third attack variant based on the cache-based approach to bypass ASLR. We evaluate countermeasures on both the cache-based and the AVX-based approach. We investigate whether the attacks work on virtualised and cloud-hosted environments. At last, we evaluate the performance and discuss the impact of these attacks.

We show under which setup NetSpectre is possible and define its requirements. Additionally, we discuss how AVX leak timing information. We demonstrate how an application can be exploited via Spectre over the network on a cache-based attack variant and an attack based on AVX. Furthermore, we demonstrate how we can bypass ASLR [78] using NetSpectre.

Motivation

Securing hardware against side-channel attacks is one of the hottest topics in IT security. At the moment, side-channel attacks are hard to mitigate and might be used more frequently in malware. Furthermore, nearly every device, such as PCs, mobile phones but also network devices, on the current market is vulnerable to Spectre. These vulnerabilities in the speculative execution were found in nearly every CPU architecture like ARM, AMD and Intel [49]. Attacking devices without system access over the network is a quite powerful attack vector.

So far, Spectre was considered to be a local attack. The primary motivation of this thesis is to investigate, whether it is possible to mount a side-channel attack like Spectre under a networking aspect. Our primary requirement is to steal secret data from a remote system without having access to the system or placing and executing malicious code on it. NetSpectre is comparable to a web server leakage attack like Heartbleed [13], where vulnerable code is used to leak sensitive data. The only attacking point is an application programming interface (API), or a program which is publicly accessible and the code of the running application is known. This attack makes it possible, without direct access to the device, to leak sensitive data from the memory of target systems. The attacker hereby mistrains the server application remotely and evaluates response times. By distinguishing the response times the attacker is capable of identifying zero and one bits. Another important requirement for this attack is a stable network connection to the target. The consequences of this attack are serious, as this attack could be used for industrial espionage. For this attack, one can imagine the following practical scenario. An online shop stores user data such as usernames, passwords and credit cards. Like almost every online shop, this offers customers a way to view their data. If other users also access their data, this data is again in the cache of the processor. With NetSpectre it is possible to leak bits of the customer data without permission by evaluating the response times as explained above. The attacker thus manages without data theft to leak sensitive data. Another scenario would be a wrongly implemented networking driver. Since drivers usually run in a higher-privileged mode, exploiting them allows leaking protected memory. The attacker would here try to utilize the driver and extract kernel memory information.

These three attack vectors give us new possibilities to use side-channel attacks and exploit it under a networking aspect. Using the gathered findings, more sophisticated approaches can be created.

Document structure

The main contributions of this thesis are:

- a proof of concept of Spectre attacks over networks
- creating a cache covert channel over the network without clflush
- breaking ASLR using weak Spectre gadgets
- porting the attack to also work with AVX instructions

The remainder of this thesis explains the necessary background explaining caches, branch prediction, Spectre and side-channel attacks in Chapter 2. The basic attack primitives are explicated in Chapter 3. The implementation and an evaluation of NetSpectre attack is given in Chapter 4 and Chapter 5. An outlook for future work and a summary is contained in the Chapters 6 and 7.

Chapter 2

Background

In this chapter, we explain the theoretical background of this thesis. We define several acronyms and terms which are used in the following chapters. Additionally, we demonstrate how timing information is leaked from caches and how those could be used to distinguish between cache hits and misses. We also explain some attack vectors and attacks based on caches.

In the first section, we explain CPU caches and its properties. We present current techniques used in cache-based side-channel attacks. Furthermore, we explain current microarchitectural attacks.

In the second part, we explain the principles of branch prediction, out-of-order exeuction and speculative execution. We discuss how a simple branch prediction unit is built and how current CPUs use these techniques to improve the performance. We explain how Spectre exploits transient execution, which allows an attacker to leak sensitive data.

At last, we explain how AVX, covert channels and Address-Space-Layout-Randomization work.

2.1 Cache

A CPU cache is used for speeding up memory accesses on the CPU. The cache is a fast, buffered memory which is located between the CPU and the main memory (RAM). A cache stores copies of data and instructions which are used regularly in the main memory.

Cache hierarchy

Typically, processors use multi-level caches, which are hierarchically ordered. The topologically closer the cache to the CPU is, the smaller and faster it is. Modern CPUs typically have 3 levels of caches [18]. A CPU typically consists of multiple cores. In most modern CPUs, each core has a private L1 and L2 cache [18, 35]. The last level cache is shared amongst the cores [18].

The first level (L1) is typically quite small. In most cases, the capacity lies between 16 and 64 KB [18]. An L1 cache is divided into a data and an instruction cache [18]. L2 caches are larger than L1 caches and can also be used for instruction and data caching [18]. The cache uses a coherence protocol to synchronise the caches of each core in order to provide data consistency. This ensures that there is always the same value stored between the different cores. Using such a protocol, the L3 cache accelerates the data transfer between cores.

Inclusiveness

An *inclusive* cache level holds all cache lines from the lower level [95]. Using inclusive cache levels, the data consistency can be more easily guaranteed, since only highest-level cache accesses need to be tracked. The disadvantage of inclusive cache levels is that memory gets wasted, due to redundant data storage. There are also *exclusive* cache levels [95]. These bring the advantage to have higher capacity within the same cache [95]. Here, a particular data is only found in one cache level and does not occur in another cache level. However, since more data has to be kept consistent, the cache coherency is more complex [95]. A *non-inclusive* cache is not explicitly inclusive or exclusive. This means there is no guarantee that data located in the upper level is also located in the lower level. If a data request occurs, the first lookup in the cache starts from the L1 cache down the hierarchy to the last-level cache. Is the data found in the cache, we speak of a **cache hit**. If the data is not found, this means that memory needs to be loaded, we speak of a **cache miss**. It is obvious that cached data is therefore accessed faster than memory that is uncached.

Replacement policies

If the cache is full, a heuristic decides what data stays in the cache and what gets replaced. This heuristic is called replacement policy. A quite simple replacement policy which is practical, is to choose the replaced entry randomly.

Another replacement policy would be for instance the *least-recently-used* policy. When the cache is full, the entry which was longest unused is replaced. This policy can be quite expensive, since the “age” of the entries has to be kept and supervised.

Mapping types

The cache can be mapped in certain ways [30]. For a *direct-mapped cache*, each address in the main memory is mapped to exactly one cache line [30]. With this approach, not all the available data in a cache line is used [30]. Caches are therefore partitioned into cache sets consisting of multiple lines. Each address is mapped to multiple lines which are grouped together in a set. However, data can be stored in an arbitrary block within the set. Setwise organised caches are called *set associative.* A cache line contains a tag, an index, the copy of the data and an offset [41]. A tag is used to distinguish different addresses in a set. The index block identifies the set number. The offset identifies a certain location within the line. If we can choose a single location out of N possible places, we call the cache an *N-way associative* [41]. We call a cache *fully associative* if the replacement policy can select an arbitrary line for the data to be stored.

Microarchitectural attacks

The Instruction Set Architecture(ISA) defines an interface between soft- and hardware. The ISA defines all instructions supported by the CPU, the memory model, execeptions, interrupts and register states in a high level. Different machines have different ISAs, for instance, x86 or ARMv8-a. If a program is defined for a certain ISA, it can run on any CPU using the same ISA. The microarchitecture refers to the underlying implementation on a transistor level of an ISA on a processor [35]. In other words, the microarchitecture details how the instructions defined by the ISA are correctly handled to achieve the expected outcome. For instance, how the microarchitectural components like pipelining, branch prediction, out-of-order execution and caches are implemented on the CPU [41].

Many microarchitectural attacks aim to expose sensitive information from hardware. Typical attack targets are CPU caches, DRAM or small cryptographic devices [28,66,69, 91]. Side-channel information like timings, the usage of memory, can be observed by an attacker and analysed. This information allows the attacker to detect certain memory accesses. With this knowledge there were several attacks built, like spying the keystrokes of libraries [28,89,91], building cross-VM covert channels [61] or attacking cryptographic implementations [66].

This thesis has a strong focus on cache attacks.

Cache attacks

Cache attacks have been improved and optimised over the last few years. The most significant advantage of caching but also the problem in terms of side-channel information is the timing difference (delta) between a cache hit and a cache miss. This delta can be exploited to identify which data was accessed and which not by an attacker.

Figure 2.1 shows the histograms of a cache hit (hatched) and a cache miss. It can be seen from the figure that there is a timing gap of 150 to 200 cycles between a hit and miss. A single cycle is for instance for 1 GHz CPU one nanosecond.

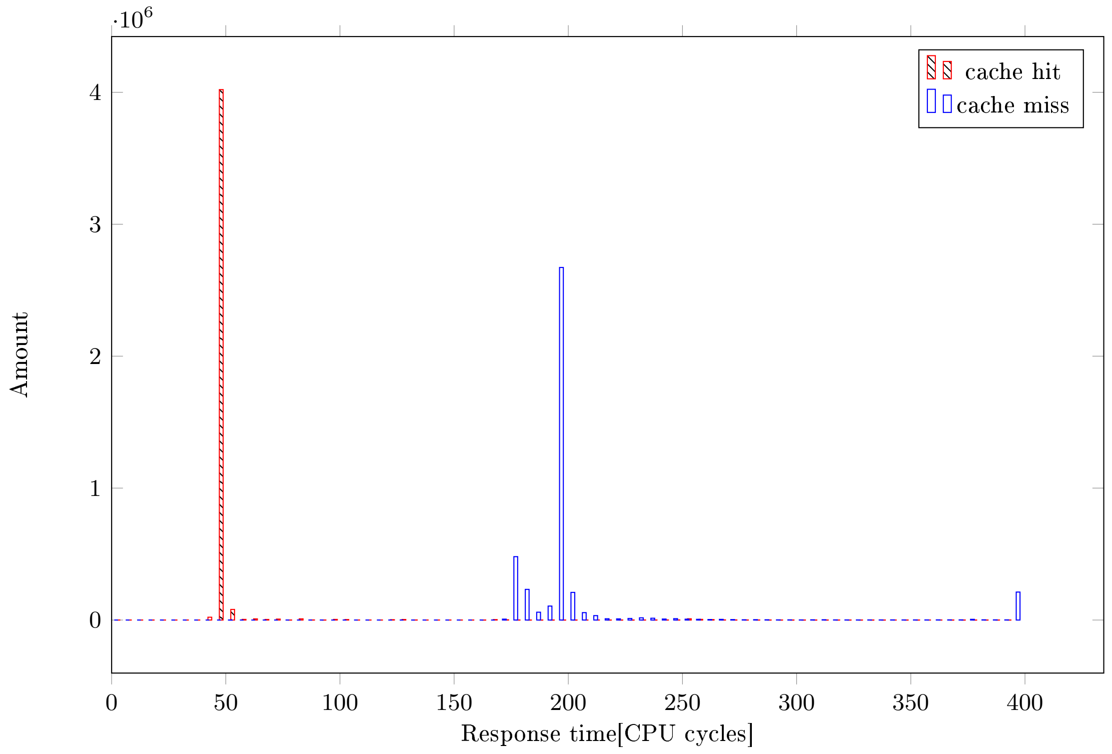


Figure 2.1: Cache timing difference between cache hits and misses on an Intel CPU.

Flush+Reload

The scenario of Flush+Reload proposed by Gullasch et al [91] is as follows. There exists a shared memory between a victim and an attacker process. If an address is accessed by either the victim or the attacker, the accessed data is cached once. Respectively, if the address is flushed, it gets flushed for both processes. The attacker is able to learn all the accesses of the victim by measuring the timing differences, when accessing a certain address. Listing 2.1 shows how to measure the timing between a cache hit and a cache miss. The necessary instructions for the Flush+Reload attack are also given in Listing 2.1. Whereas **rdtsc** returns the current timestamp of the CPU, **maccess** access a memory location at a particular address and **flush** triggers the clflush instruction to flush every occurrence of a memory address in a certain cache line [35]. Gruss et al. [28] automated these types of attacks and were capable of detecting keystrokes of shared libraries. Flush+Reload is used in transient execution attacks like Spectre, Meltdown and Foreshadow to identify memory accesses and therefore leak data [83].

Other cache attacks

If the *clflush* instruction is not available, for instance on ARMv7 [5], another attacking strategy was developed. Gruss et al. [28] developed an attack called *Evict+Reload*. In order to flush (evict) a certain address out of the cache, many addresses that are congruent, regarding to the replacement policy, need to be accessed. If enough such addresses are accessed, the specific address is evicted from of the cache. After the eviction part, the victim's process is scheduled, which accesses certain cache line. The attacker then accesses the cache lines and checks the access times for cache hits and misses. For simple cache replacement policies like LRU, it is easy to find an appropriate eviction strategy [28]. However, for random replacement this can be a hard task since the implementation is unknown. Cache eviction strategies are crucial, since for NetSpectre attack we later on also need to evict certain addresses out of the cache [28].

Gruss et al. also showed a new attack only based on flushing of cached and uncached addresses [26]. This method was called *Flush+Flush*. Again, there is a timing difference when flushing cached and uncached memory. This attack has the same requirements as Flush+Reload. There are also cache attacks which do not need shared memory. A technique is called *Prime+Probe* [66]. For Prime+Probe, it is necessary to reverse engineer the used cache replacement policy. Using this knowledge, the attacker tries to fill specific cache sets. Then the attacker waits until the target application runs. The application fills used cache sets. Afterwards, the attacker verifies which of the cache sets are still filled by measuring the access time. This is done, by reaccessing all pages again and distinguishing between access times. Slower access times imply an access of the victim's process, since the attacker's address was evicted by the victim.

```
uint64_t rdtsc()
{
  uint64_t a, d;
  asm volatile ("mfence");
  asm volatile ("rdtsc" : "=a" (a), "=d" (d));
  a = (d<<32) | a;
  asm volatile ("mfence");
  return a;
}
void maccess(void* p)
{
  asm volatile ("movq␣(%0),␣%%rax\n"
    :
    : "c" (p)
    : "rax");
}
void flush(void* p)
{
    asm volatile ("clflush␣0(%0)\n"
      :
      : "c" (p)
      : "rax");
}

#define CACHE_MISS 185
int flush_reload(void* ptr)
{
  uint64_t start = 0, end = 0;
  start = rdtsc();
  maccess(ptr);
  end = rdtsc();
  flush(ptr);
  if(end - start < CACHE_MISS)
  {
    return 1;
  }
  return 0;
}
```

Listing 2.1: Definition of rdtsc, memory access (maccess) and flush.

2.2 Pipelining and Branch Prediction

Instruction pipelining was introduced to increase the efficiency of the processor. The main tasks of a CPU are as follows. First the instruction gets fetched, afterwards the instruction gets decoded, and at last, the instruction gets executed [18].

Since the instructions are executed in parallel, problems can occur which are called hazards [12]. There exist three types of hazards which need to be resolved properly [41]:

- Data hazards occur if an instruction is fetched and the next instruction has a data dependency on the previous instruction.
- Structural hazards occurs if a resource is needed by multiple instructions. I.e., an instruction that loads a certain memory address and another instruction that writes to this address creates a conflict.
- Control hazards occur from branch instructions that influence the program counter

Stalling is the delay in a pipeline which resolves a hazard [41]. In modern CPUs, the stalling time should be kept low to guarantee the perfomance. Data hazards can be resolved by pipeline gaps (bubbles), out-of-order execution or operand forwarding [41]. Structural hazards can be resolved by using a separate cache for instructions and data [41].

The solution to resolve control hazards is branch prediction. If instructions after a branch get fetched and evaluated, but the branch is not taken, these instructions need to get invalidated. **Flushing** is the time when clearing these unnecessarily executed instructions from the pipeline and starting the pipeline from the new correct instruction. These instructions are executed *in-order*. In order to reduce stalling and flushing times, branch prediction was introduced.

A **branch predictor** takes the current state of a program (program counter) as an input and tries to predict the next instructions for a given branch [41]. The problem which needs then to be solved for conditional branches is the *direction* of a branch. Direction means, whether the branch is taken or not [41]. For indirect branches there often exist more than one target or the target might be known at runtime [68]. In the following subsections, we discuss static and dynamic branch prediction to resolve those problems.

Static prediction

In *static* prediction, the guesses are created at compile time [80]. A table is generated beforehand and does not change at runtime. Static prediction is used in cases where the outcome of a branch is highly predictable.

Predict Single direction

The easiest strategy is to predict that all branches always have the same direction [43]. Only taken or not taken branches are considered. For taken branches, the target address has to be computed early. If this not possible, the CPU stalls. If the prediction only considers not taken branches, there occurs a delay if the prediction was wrong.

Backward Taken Forward Not Taken

This strategy can also be used to predict branches. As the name says, backward branches, i.e. branches, where the address is lower than the address of the branch, are taken [43]. For instance, in loops, the prediction accuracy here is quite high, since the branch is usually at the bottom of the code.

Profile-based approach

Based on profiles taken from previous runs with sample input, the branch prediction is fed and statically generates a table. For instance, compiler hints can be set to improve the prediction results [43].

Static prediction is easy to implement in hardware. The biggest drawback is the static decision, which means once predicted the prediction stays. However, the above mentioned drawbacks are too costly, a more dynamic approach is required. Static branch prediction can be used in combination with dynamic branch prediction. For instance, if the dynamic branch prediction unit is overloaded, static branch prediction could be used as fallback.

Dynamic prediction

Dynamic prediction relies on different and more sophisticated techniques to guess the branch target [80]. When executing a program, the branch prediction learns at runtime branch targets and the direction of branches for each execution. No preprocessing or profiling is necessary for dynamic prediction. Dynamic branch prediction first tries to predict whether a particular branch is taken or not taken. Then, based on this result, the target address of the branch gets predicted.

Branch History Table

A Branch History Table (BHT) is a table which contains at least a single bit for each branch, which indicates whether the branch was previously taken or not taken. Depending on the value of the bit, the branch is predicted as taken(1) or not taken(0). This table is ordered by the first *k* least significant bits of the address from the branch. Figure 2.2 illustrates a BHT, where the first k bits are selected to identify a branch by its address and not taken.

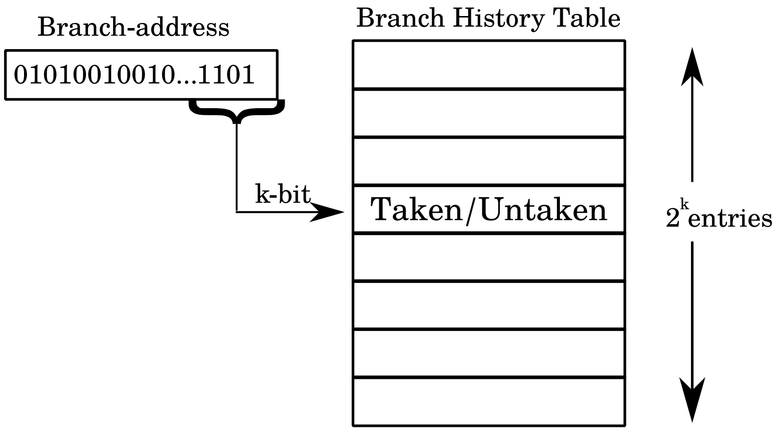


Figure 2.2: Illustration of Branch History Table.

Using a one-bit BHT, two mispredictions occur when using branches that are always taken as static prediction for loops [41]. In the last iteration, the one bit BHT again predicts the branch, however, it should exit the loop. The branch predictor switches its state to not taken. In the next execution of the branch, the prediction is not taken. For instance, for 10 iterations the accuracy is only 80 percent. To increase the accuracy, the branch history table is extended to two or more bits. Figure 2.3 illustrates a one bit predictor.

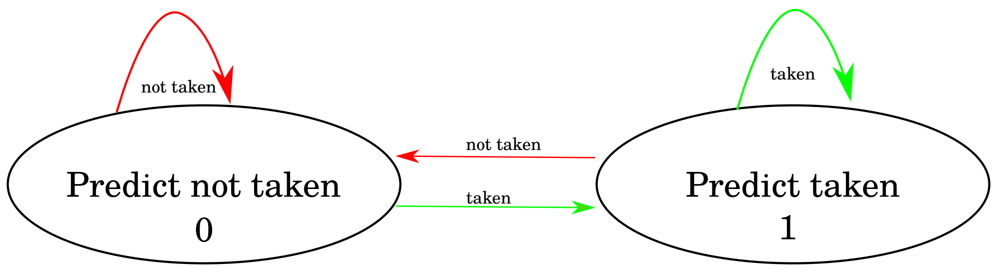


Figure 2.3: State machine of a one-bit branch predictor. [41]

For a two-bit predictor, the BHT contains entries with two bits. Figure 2.4 illustrates the state machine of a two-bit predictor. These two-bit predictors can also be seen as counters [41]. In the first run the initial state is 00. If the branch is taken in the first case the last bit flips to 01. If not, the state 00 stays. If in the state 01 the condition is once again correctly predicted, the state changes to 10. As can be seen, if the branch is two times not taken, the prediction switches from taken two not taken. Conversely, if the branch is two times taken from not taken state 00, it predicts taken on the next run.

In the loop example from before, the branch accuracy is improved since only one misprediction occurs in the last iteration. That is because the two-bit branch predictor is in the state 11 and switches to state 10. Thus, in the next execution of the loop, the branch is again be taken. This two-bit predictor can be easily extended to an n-bit predictor. It is like adding or subtracting a certain number by changing a single bit. As Yen et al. [92] showed, the two-bit predictor is the most effective predictor.

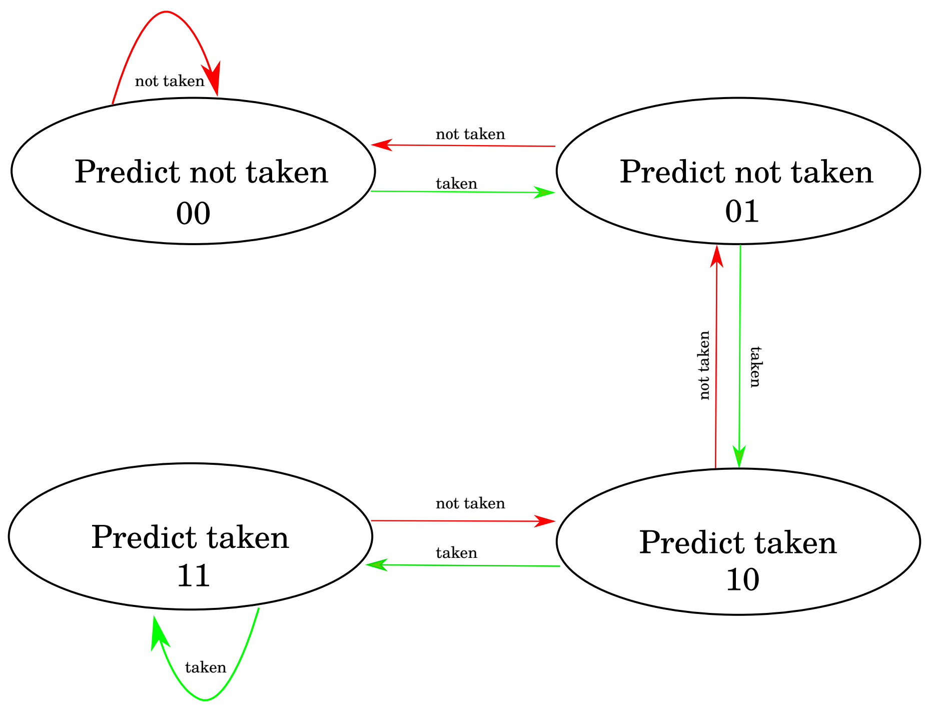


Figure 2.4: State machine of a two-bit BHT state machine.

Yen et al. furthermore proposed to use a two-level branch prediction [92]. The first level is a branch history register or directly the branch history table. The second level is again a table called Pattern History Table (PHT) [43], which is a table consisting of four two-bit predictors per branch. Depending on which value the branch in the BHT has, one of those four counters is chosen.

From this base, different techniques were developed which are not discussed further [43]:

- Index-Sharing Predictors
- Interference Reducing Predictors
- The Bi-Mode Predictor
- Variable Path Length Predictors
- Perceptron Predictors

Target prediction

For a conditional branch also the target address to jump after predicting whether this branch is taken or not needs to be predicted. The Branch Target Buffer (BTB) stores the predicted results and their corresponding source addresses [68]. If an instruction is fetched it is looked up in the BTB beforehand. Furthermore, it optionally stores whether the predicted branch was taken or not [41]. Figure 2.5 illustrates this lookup via a Branch Target Buffer. The first column contains a set of all known branch addresses. The second column contains the prediction for this branch. If the branch was not taken, the prediction is the following instruction. Conversely, if the branch is predicted to be taken, the predicted target is the last target, where the branch jumped. If the prediction is wrong, the CPU stalls for the next instruction. The BTB can be combined with the branch prediction from before to determine the target address [68].

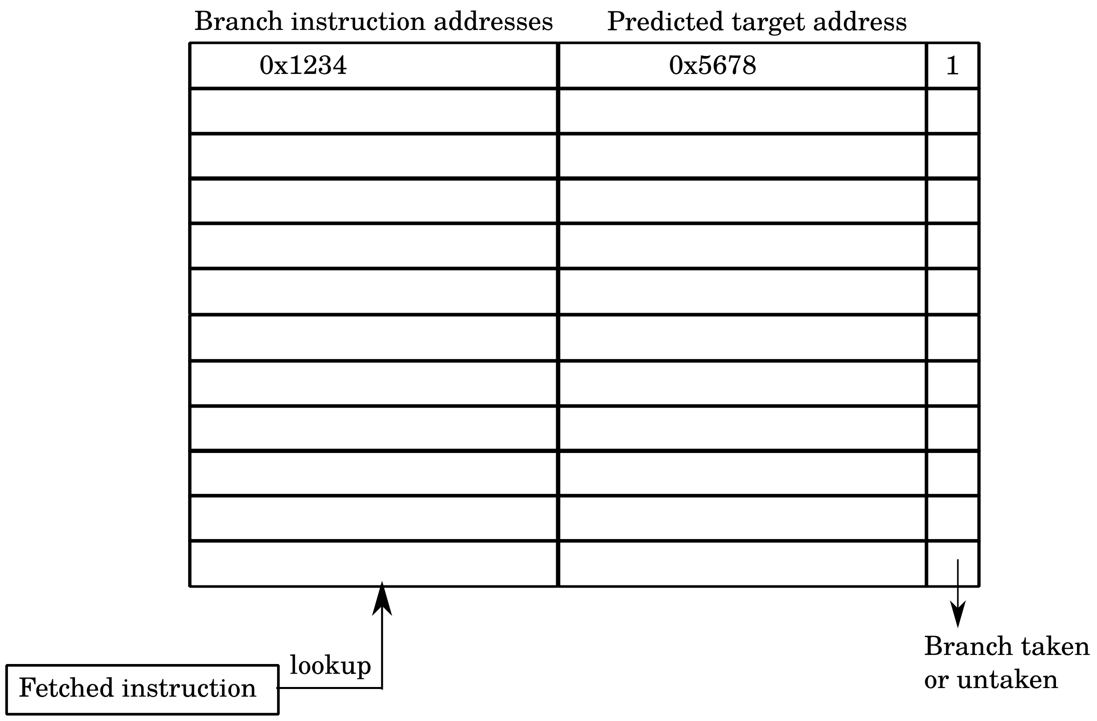


Figure 2.5: Illustration of Branch Target Buffer [41]. The virtual address of the instruction (i.e. value of the program counter) is looked up in the first column. If there exists a predicted target, the predicted result is taken. If not, the PC proceeds normally.

Return-stack buffer

Additionally to the target address, the return address needs to be predicted. To solve this problem return-stack buffers got introduced. A *return* instruction is comparable to an indirect branch [43]. Thus, the target of the instruction can also be predicted and speculatively executed. For this case, a special hardware buffer called Return Stack Buffer (RSB) was invented. The RSB contains the most current return addresses [41]. If a call instruction gets executed, the predicted return address gets pushed on the RSB. Conversely, if a return instruction is executed, the return address is popped off the stack. The BTB marks all return instructions with a special flag [68]. When the return instruction gets predicted, the top address of the RSB is taken, instead of the mapped target address. The RSB is typically contains 16 entries which appear to be enough to reduce misprediction to nearly zero [41].

Out-Of-Order execution

Out-of-Order (OoO) execution was introduced to overcome CPU stalling problems and to increase the overall performance of CPUs [41]. An OoO-CPU dynamically schedules instructions [41]. Here, after the decoding phase of instruction checks, for structural and data hazards are performed [41]. With this knowledge, instructions that are not dependent can be executed in parallel to reduce the idle time. The instructions are completed out-of-order. However, before the results are visible to in architectural state, the instructions are reordered in-order. For the reordering, a special buffer, namely reorder buffer (ROB), is used. The buffer stores and updates the indermediate results of the instructions in the buffer. When the execution is commited, the registers are updated [41]. However the microarchitectural state, i.e. changes in the cache, is not reverted. Instructions that are not commited but change the microarchitectural state are called *transient* instructions [54]. Listing 2.2 illustrates a simple example where out-of-order execution can be used to increase the performance. Here, the add instruction cannot be executed since it depends on the previous memory load. However, the mov instruction can be executed in parallel since it does not depend on the previous instructions.

```
mov eax,0x539
add eax,9
mov ebx,5
```

Listing 2.2: Example for out-of-order execution.

Speculative execution

Speculative execution is used to speed up the performance of pipelined processors. This approach now combines the concept of branch prediction and out-of-order execution [41].

Speculative execution that relies on predictions is called *predictive* speculative execution. According to Ge et al. [20], the BTB and RSB results are only shared within the same core. In the decoding phase, a branch instruction can be identified. It uses the predicted results of branch predictors to pre execute the predicted branch. If a conditional branch instruction is fetched by the CPU the current architectural state is stored. Possible execution paths are predicted by the branch predictor and then executed speculatively. Afterwards, when the branch instruction is executed, the result is validated. If the prediction was correct, the execution resumes from the predicted path and discards the stored state. Otherwise, the results of the execution are discarded and the stored state is restored. From this restored state, the other part of the branch is executed. However, transient instructions were executed which changed the microarchitectural state.

2.3 Transient execution attacks

There were earlier attacks on the components of the branch prediction unit attacking cryptographic primitives [63]. Speculatively accessed data is still accessible for instance in the cache, which makes it vulnerable to perform transient execution attacks. In January 2018, Kocher et al. [49] showed a practical attack named Spectre on exploiting the speculative execution unit. This attack is both possible using native code and also via JavaScript. Spectre allows an attacker to leak sensitive information via a cache side-channel attack (e.g. Flush+Reload).

Spectre V1

Spectre can be used to exploit conditional branches. We assume a simple example like in Listing 2.3. The attacker has control over a variable *vuln*, which is an index of an array. If the condition is true, the index is lower than the allowed maximum; memory is accessed and therefore lands in the cache. If the condition is false, e.g., a too large index is given, then the memory access should not be taken. In the previous section, we explained that predicted branches are executed speculatively. Thus, in order to exploit this conditional branch an attacker would first input valid indices to mistrain the branch prediction. On average, 5-20 attempts are enough to mistrain the predictor [49]. The branch taken bit of the PHT is set to true and afterwards the branch is executed speculatively [8]. The branch prediction learns that the branch is probably taken and via speculative execution the data is accessed at the offset. After mistraining, the attacker sends an out-of-bounds index, which causes that the valid branch is not executed and the data not accessed in the architectural state. However, the mistrained branch predictor executes the branch speculatively and changes the microarchitectural state, i.e., the data stays in the cache. In Listing 2.3, the attacker would, for instance, send 10 times the index 1 to mistrain the branch prediction. Afterwards, the attacker sends one out-of-bounds, e.g.,4. The attacker performs Flush+Reload on the same address [91]. Since the attacker does not know which byte was accessed, the address tries all values from 0 to 255 and measures the access time. If the time is below the cache threshold, in this example for `mem[data[4]]`, the attacker leaked the value 'S'. The attacker could repeat this attack to leak arbitrary memory accessible for this application. This type of attack is called Spectre V1(Bounds check bypass on Loads) [49].

Spectre V1.1 (Bounds check bypass on stores) [47] is like Spectre V1, with the difference that instead of loading data a store operation like "mem[data[vuln]] = value".

```c
unsigned char* data = "dataSECRET";
char* mem;
char access_array(int vuln)
{
  flush(&vuln);
  flush(data);

  if(vuln < strlen(data) - strlen("SECRET"))
  {
    return mem[data[vuln] * 4096];
  }
}
```

Listing 2.3: Conditional branch which is vulnerable to Spectre V1.

Spectre V2 and other variants

In indirect branches, the target location where to jump to depends on a value of a register, e.g., RAX or the value of a memory location Before execution, the jump's target address is not known. The usage of indirect branches is, for example, if a specific part of the code needs to be executed depending on a certain input.

Attacking indirect branches can be compared to a well-known exploitation technique called Return-Oriented-Programming [72] (ROP). In ROP, the attacker tries to use code snippets to jump from one snippet to another by overwriting the return address. For instance, in a buffer-overflow attack, ROP is used to leak information and bypass system hardening mechanisms such as ASLR [78].These small snippets are called **gadgets**. Using this possibility, code pieces could be again "sticked" together, and be misused for malicious purposes. Inspired by Return-Oriented-Programming [72], the discoverer of Spectre call parts of the code which are vulnerable to mistraining also **gadgets**. Those gadgets could the be used to create side-channel information [49]. We also use this term for later defining the necessary code snippets we need for performing NetSpectre.

In `Spectre V2` (Spectre-BTB) attack, the attacker tries to inject any target address to the BTB [8, 49]. For Spectre V2, the following scenario can be considered. The attacker has knowledge of the virtual address of a gadget in the victim's userspace which is, for example, the address of a function which again accesses data relatively to an index. In other words calling a Spectre V1 branch. The attacker runs a process which constantly calls the address of the Spectre gadget. The victim process also contains an indirect branch, which is mispredicted after mistraining. The gadget accesses a certain memory location which can be leaked via a cache-side channel [49].

Another example for exploiting Spectre V2 is by using polymorphism in C++ with two classes. Both classes contain a function, where one is accessible for the attacker. The other class contains a Spectre gadget. Here, the attacker would call the accessible function multiple times to mistrain the branch prediction. Afterwards, the attacker would inject into the branch predictor to a known virtual address of the Spectre gadget. This function is speculatively executed and the execution has again side effects on the microarchitectural state. Like the Spectre V1 on conditional branches, the results of the mistraining can be exploited via Flush+Reload.

This attack can be used to leak host-memory from a virtualized guest [49]. Further Spectre variants [36, 52] were released. The attack was ported to Intel's SGX by Chen et. al. [10]. Intel SGX (Software Guard Extensions or SGX) is an extension set of secure instructions [10]. Here an enclave is used that encrypts and protectss code and data from being spied and tampered.

Store buffers are used to improve the performance of memory writes [8]. Spectre V4 (Speculative Store Bypass) exploits the store-to-forward logic in store buffers [8]. The Speculative Store Bypass is an optimization technique which allows that a load instruction may be executed speculatively depending on an older store value [8]. The attack can be compared to winning a race condition [8]. If the load is mispredicted to be non-dependent on a previous performed store instruction, it will be executed speculatively. However, this misprediction changes the microarchitectural state as the load instruction accesses memory depending on an unsanitzed value [8]. This memory access can be again evaluated using Flush+Reload.

Spectre V5 (ret2spec) exploits speculative execution via the usage of Return Stack Buffers [58]. Furthermore, Evtyushkin et al. [17] introduced a new side channel attack called Branchscope, which attacks the directional branch predictor.

Meltdown

Meltdown allows an attacker to read arbitrary memory without any necessary privileges [54]. The only requirement to perform Meltdown is code execution on the victim's machine. Meltdown does not rely on exploiting the branch predictor [54]. Here, the attacker directly accesses memory that should be inaccessible. This access causes a fault which throws an exception. However, if a certain trap is reached due to an instruction, the following instructions are nevertheless executed out of order. This allows an attacker to efficiently leak the entire content of the RAM [54] via the cache accesses. Listing [54] illustrates a proof-of-concept. First, a null pointer gets derefenced and then data which should not be accessible is accessed. The program should crash immediately and if the exception is handled, the memory access can be validated via Flush+Reload. Meltdown

```c
*(volatile char*)NULL;
mem[vul * 4096] = 0;
```

Listing 2.4: Meltdown example access that gets performed due to out-of-order execution

got patched on Linux via the KAISER patch, which unmaps the kernelspace whenever userspace operations are performed [24]. Spectre V1.2 allows to speculatively overwrite read-only memory [47]. The name is misleading, since Spectre V1.2 is a Meltdown variant [8].

Van Bulck et al. [83] presented a new attack called `Foreshadow` first performed on SGX. This attack enables an attacker to steal sensitive data from secure enclaves and hypervisors. Since vanilla Meltdown cannot be used in SGX enclaves, Foreshadow clears the present bit in the page-table entry of SGX to ensure that a pagefault happens [8]. If the present bit is cleared, an L1 lookup with the virtual address happens directly without permission checks [83]. Foreshadow NG [88] generalizes this approach to bypass isolation from guest to host in virtualization.

2.4 SIMD Instructions

Single instruction multiple data (SIMD) is a concept that is designed to execute a certain operation on numerous data points [33]. The big advantage of this concept is the performance. For instance, for matrix operations like an addition with a scalar, the addition is applied to the whole data block.

Advanced Vector Instructions (AVX) extend the standard instruction set of the Intel x86 architecture [33]. These instructions can be used to improve the performance of programs which use vector operations. An application of AVX instructions is the accelerations of cryptographic primitives. For instance, common block ciphers like AES can be implemented using AVX instructions [33]. These operations are commonly used in computer graphics. Similar behaviour to cache hits and misses got observed when measuring the timing of AVX instructions [19]. This behaviour and how to exploit it, is explained in Chapter 3.

2.5 Covert Channels

In a covert channel, two processes want to communicate with each other, even though they are not permitted to. In most cases, the two processes use a shared medium, which they are both allowed to access [87]. This could be, for instance, Flush+Reload on a shared memory [91]. A shared memory could be for instance a library or simply a memory region that is used by multiple processes. By sharing the memory to multiple parties, memory is saved. Sender and the receiver use a specific amount of addresses from a shared memory object [91]. In order to transmit bitwise, a fast memory access (e.g. cache hit) represents a one and a slow access represents a zero (e.g. cache miss). To increase the capacity of the channel, it is common to use addresses that map into different cache sets. The performance of covert channels is the crucial property. There have been covert channels developed for TCP/IP implementations [55]. For Android, also sensor-based covert channels were built, which could be used for spying on the user [2]. It has been shown that these covert channels also work on cloud-hosted environments [69,89]. Research also showed covert channels via Interrupts [59], DRAM [69], branch predictors [15], memory buses [74] or the dynamic frequency scaling of the CPU [3].

To measure the performance of covert channels, the true channel capacity is used for a binary symmetric channel [11].

$$Cap = RC * (1 + (p_{err} * log_2(p_{err}) + (1 - p_{err}) * log_2(1 - p_{err})))$$

RC describes the raw channel capacity, meaning which amount of bits are noiseless possible. p_{err} describes the probability of an incorrect bit. The unit is measured in bits per second. Current cache-based covert channels achieve a true channel capacity of a few Megabits per second [26].

2.6 Virtual Memory Separation

Modern operating systems like Windows or Linux split up its virtual memory into userspace and kernelspace. The kernelspace is basically the memory region where the operating systems runs and provides its applications and services. All user programs run in a separate memory location called userspace. Userspace applications are restricted in such a way that they are not privileged to access the kernelspace. The userspace can access the kernel only through system calls.

2.7 Address Space Layout Randomization

The main aim of Address Space Layout Randomization (ASLR) is to randomize virtual addresses in the memory [78]. This means on each program execution, data is located at a different location. Randomized memory regions could be the stack, the heap, data sections and the BSS segments. Depending on which compile flags are set, not all of the above mentioned regions are randomized. ASLR complicates memory corruption attacks like buffer overflows. This is because of the fact that an attacker has to leak specific addresses to calculate the randomized offset. If the attacker is capable of leaking one address, he is able to calculate back the offset and exploit the program. If there is no possibility to leak addresses, the attacker has to guess the address. The lower 12 bits of each address are fixed since the addresses have to be page-aligned. This leaves for 32-bit virtual addresses 20 bits to randomize. One additional bit is needed to distinguish between the memory regions. Which leaves 19 bits entropy and 2^{19} guesses have to be performed [78]. On 64-bit architecture the effort is higher since the addresses are larger and there is more space left for randomization. Under a 64-bit architecture, the number of bits to guess is 28 and on the PAX implementation it is 40 [60]. Research also showed different side-channel attacks to bypass kernelspace ASLR [16, 25, 31, 39] ASLR could be bypassed using more advanced attack techniques like Return-Oriented-Programming [72].

Chapter 3

Attack Primitives

In this chapter, we explain the attacking primitives we need for NetSpectre. We show how to use the primitives to create timing differences and how to measure them. We want to detect a possibility to mistrain gadgets of server-side code. As a proof of concept for networking, we consider a client-server scenario. The server part is like a Web API that is publicly available. For instance, the API provides data which is publicly accessible for specific users.

At first we explain the attack setup and give a simple attacking scenario. To create this attack, we define the necessary building blocks for [77] to perform NetSpectre.

Secondly, we discuss possible attack vectors in userspace and kernelspace. We describe where NetSpectre gadgets could be possible exist.

Thirdly, we demonstrate how to measure the timing difference on Intel CPUs. Moreover, we argue why it is necessary to repeat the measurements to increase the confidence.

At fourth, we define the necessary primitives for a cache-based attack. Additionally, we show that a simple threshold-based approach is enough to distinguish between a zero and one bit in a binary stream. We also demonstrate a possibility to flush the cache without having the ability to use a flush instruction.

At last, we show how to derive a timing difference in the execution of AVX instructions. We demonstrate how to use this difference to be able to create the first AVX-based covert channel. Furthermore, we define the necessary gadgets for an AVX-based attack.

3.1 Attack setup

Our focused Spectre variant is Spectre V1, since it is the most realistic and easiest variant to exploit [77]. Furthermore, it is very likely that a large code base contains Spectre

V1 gadgets [49]. In the classical Spectre V1 attack, the attacker mistrains a conditional branch, for example an out-of-bounds check. We discuss other Spectre variants in Chapter 4. After the mistraining phase, an out-of-bounds value will get accessed speculatively. This speculative execution causes a change in the microarchitectural state e.g. in the cache, which leads to faster execution times. The process then leaks information transmitted by a covert channel to the attacker.

In the case of NetSpectre, we modify this attack as follows. The attacking scenario is a public API where the attacker has access to a public part of a data stream. The clients are capable of receiving certain parts of the data via an index sent from the client to the server via network packets. For simplicity of this attack we consider the data to be in a binary format. For a bitwise representation, we only need to distinguish between a zero or one, respectively whether data is accessed or not. However, the accessed data can also be a byte or multiple bytes. The client (attacker) has control over an index which is sent to the API. The index is checked in a conditional branch for out-of-bounds values. The API always responds to the server after accessing or not accessing the data. The conditional branch and the attacker-controlled index form the first necessary gadget which is called `leak gadget`. Listing 3.1 illustrates a leak gadget.

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

Listing 3.1: Cache leak gadget: Conditional branch that performs out-of-bounds accesses via mistraining. The out-of-bounds access of a single bit is encoded by accessing the flag variable. [77]

The attacker sends multiple valid indices to mistrain the branch predictor. The branch predictor learns that the condition of the branch is with a high probability fulfilled. If the conditional branch gets again executed, the branch predictor will speculatively execute the part of the branch that accesses the flag variable. Thus, after the mistraining, the attacker sends one out-of-bounds index that will access the flag variable. This memory access needs to be evaluated using a second gadget, the `transmit gadget`. The API needs to provide another code snippet to validate the microarchitectural state, like Flush+Reload for Spectre V1. In our scenario, the API contains another public function that again accesses the flag variable and responds to the client. The attacker measures the response time of the received response packet. The response time of the packet should be significantly faster, if the flag variable was loaded from the cache caused by the speculation before. If this is this is the case, a 1-bit in the secret part was accessed. Conversely, if the indexed bit is a 0-bit, the response time is significantly slower. Figure 3.1 illustrates a general NetSpectre attacking scenario explained above.

This gadget allows an attacker to evaluate the speculative execution part of the attack by reaccessing the certain bit or executing a single additional instruction. The attacker uses the leak gadget and measures the response time. Beforehand, the attacker has to measure the response times for a zero bit and one bit and the public part of the API.

Using these two gadgets we are now able to build a cache- and AVX-based covert channel.

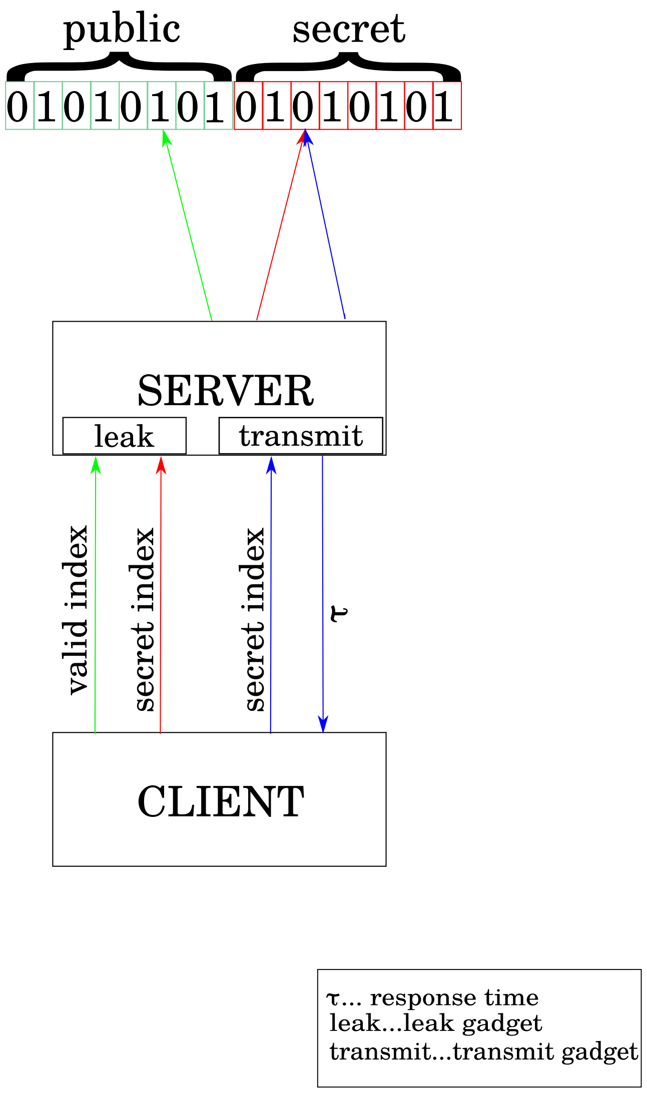


Figure 3.1: Illustration of NetSpectre attack via client-server scenario. The attacker first uses the leak gadget. First, the client sends valid indices to mistrain the branch prediction. Afterwards, the client sends at least an invalid index. Using the transmit gadget the client validates whether the accessed index was a zero or one by measuring the response times.

3.2 Attack Vectors

As already mentioned in Chapter 2, the virtual memory is divided into userspace and kernelspace. The kernelspace consists of all virtual memory necessary for the operating system. Userspace applications are not permitted to read or write kernelspace memory. The previously defined gadgets can occur in both locations.

In the userspace, we can think of a web API that receives data from an arbitrary client and responds. For the indexing scenario, we can assume that the client has the possibility to choose a particular element from a public resource via an index. In the userspace, the impact is, that arbitrary memory from userspace applications can be leaked. Potential attack targets could be HTTP servers, FTP servers, SSH servers. Leaking some information on this memory location can be used for further attacks. API keys, user credentials or sensitive user data could be leaked. Basically, any secret that is stored in the running application can be leaked using NetSpectre.

In the kernelspace, we can think of a wrongly implemented networking driver. The driver sends and receives a lot of different data. Drivers are designed to support different protocols like the internet protocol. If the two gadgets are located in a network driver, the attacker is capable of leaking arbitrary system memory.

3.3 Channel and Measurement

We want a bitwise leakage of the binary data. Thus, when using the transmit gadget after mistraining, responding network packets of a zero-bit are significantly slower than packets with a one-bit. As defined in Listing 2.1, we use the **rdtsc** instruction to measure the time of response packets.

A lot of network noise can occur, and we need to repeat the measurements multiple times to get a particular confidence for a zero- or one-bit. The distribution for the network response timings is unknown. However, we assume the data to be normal or log-normal distributed. As the Law Of Large Numbers states [75], on a long run, the average of the results converges to its expected value. We use this fact and later compare the maximum peaks (average values) to distinguish between zeros and ones. Thus, it should be sufficient to build a histogram from the response times to distinguish between a zero- and a one-bit.

3.4 Cache Primitives

In Chapter 2, we discussed the Flush+Reload respectively the Evict+Reload attack. The possibility to use the flush instruction is not available in our attacking scenario via the public API. Thus, we have to adapt Evict+Reload. The attacker tries to access a specific address, in our case a certain bit. With Evict+Reload, the attacker tries to evict a certain address out of the cache. Then the victim accesses memory and by again accessing the same virtual addresses and measuring the access times, the attacker receives knowledge about accessed memory.

For our attack, we consider data to be in its binary representation. For the proof-of-concept, it is sufficient to distinguish between a zero and one. Regarding the cache-based attack, we need to distinguish between a cache hit and a cache miss. In Evict+Reload one process would access an address in a shared memory and in another process, the same address is again accessed. In the second process, the response time is measured.

This kind of attack can be adapted to a networking side. The server creates an array and encodes the data in binary. At first, the client sends valid indices, which will be accessed by the leak gadget repeatedly. We discovered that ten to twenty valid indices are sufficient to mistrain the branch predictor on our evaluated CPUs. If the sent index relates to a one-bit, the server accesses the data. Conversely, if the sent index is a zero bit, the server does not access the data. To verify if the bit was really a one-bit, the client uses the transmit gadget and checks the response timing. If the address was accessed on the first time the response should be faster (below a treshold = cache hit response), since a cache hit occured, and therefore the index is a one-bit. Conversely, if the address was not accessed, a cache miss occured and the data was loaded from the memory. Therefore the response time should be slower (above a threshold = cache miss response) and the index is encoded as zero-bit. With this client-server cache approach the bits can be bitwise leaked.

Thus, in our attack, the attackers send twenty valid indices to mistrain the branch predictor. Afterwards, the attacker sends four out-of-bounds indices of 1 byte, which will be speculatively accessed. There could also be more than four indices be accessed depending on the number of mistrained requests before. However, we observed the best performance using this metric.

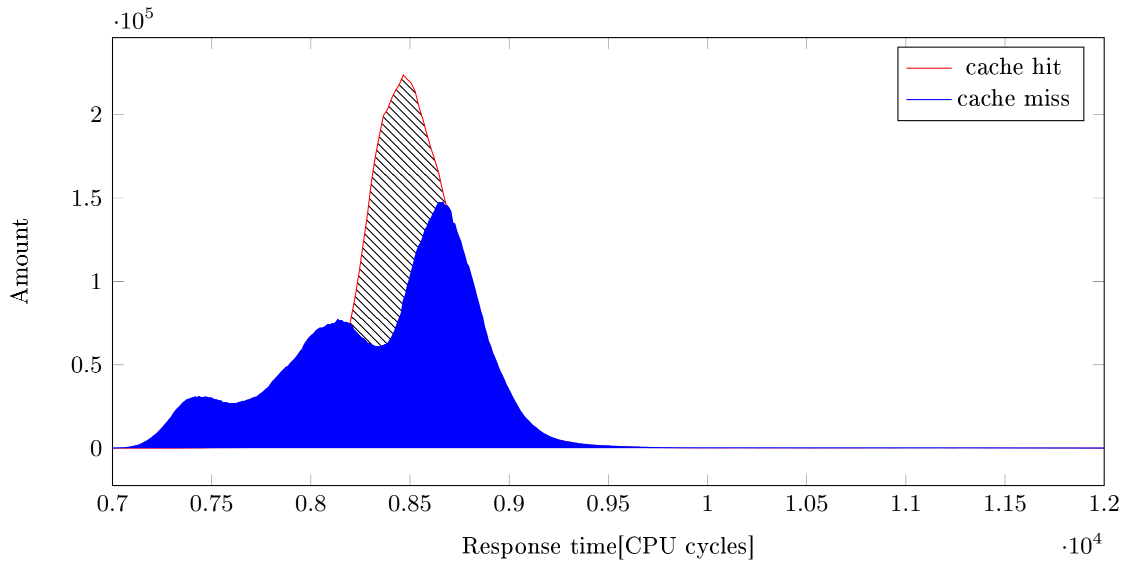
At last the client resets the testing conditions since the accessed bit needs to be evicted out of the cache.

Timing Measurement

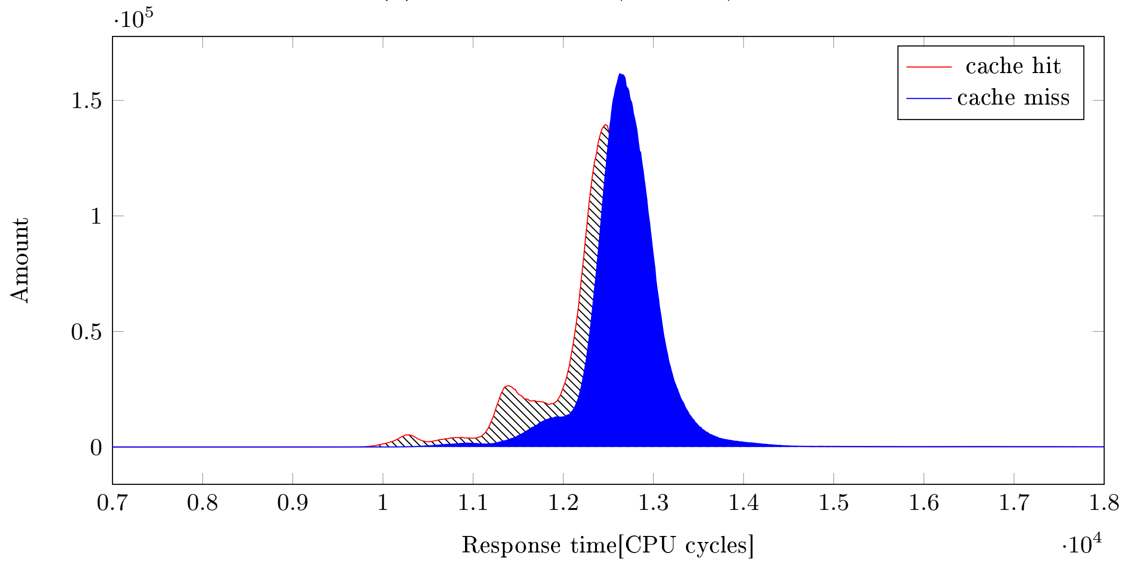
To distinguish between a cache hit response and a cache miss response, we create two histograms on the client side measuring the servers response times. The first histogram measures the timing responses for a cache miss. After analysing the histogram we are capable of defining a certain threshold to distinguish between a zero and one . Depending on the hardware and the number of network devices between the server and client, the response times will strongly differ. Thus, it is not possible to choose a general threshold to distinguish between a zero and one for all devices.

For our proof-of-concept implementation we chose plain UDP (User Datagram Protocol) sockets [65] as protocol to communicate between client and server. This client-server implementation was used to create the histograms.

We created a server-client based histogram tool, which measures the timing of an accessed bit (cache hit) and a not accessed bit (cache miss). We created histograms for both a notebook and a desktop CPU. It can bee seen from Figure 3.2 that there is a lot of noise coming from network communication. For the notebook CPU it can be seen, that in some cases, cache misses appear to be faster. These apparently faster timings come from the UDP sockets. However, on the long run (with sufficient repetitions), the timings of a cache hit are still at around 100 to 150 cycles faster on average than cache misses.



(a) Notebook CPU (i7-8550U)



(b) Desktop CPU (i7-6700K)

Figure 3.2: Timing difference between a cache hit and a cache miss over the network in a local environment for both desktop and notebook CPU.

Resetting the microarchitectural state

The second access by the transmit gadget caused a change in the microarchitectural state. In order to repeat the measurement, the architectural state has to be resetted. Notably, the address has to be evicted from the cache. Since a direct possibility to flush the address is not possible, an alternative has to be found. We need to adapt the *Evict+Reload* attack. Instead of *Evict+Reload*, we call this strategy to remotely evict *Thrash+Reload* [77]. Thus, we have to find a specific eviction strategy in order to evict the accessed bits out of the cache. We discovered that a constant file download on the server is enough to create a similar behaviour like evicting [77]. Depending on the cache size of the CPU, the file size varies. An Intel i5-6200U (3 MB cache) a file download of around 1 MB is enough to evict a single variable. For an Intel Core i7-8550U with 8 MB large last-level cache a file download of around 8 MB is necessary to completely evict the variable.

Figure 3.3 illustrates the probability of a local variable being evicted from the cache.

Using the above mentioned gadgets, we should be capable of building a cache-based attack.

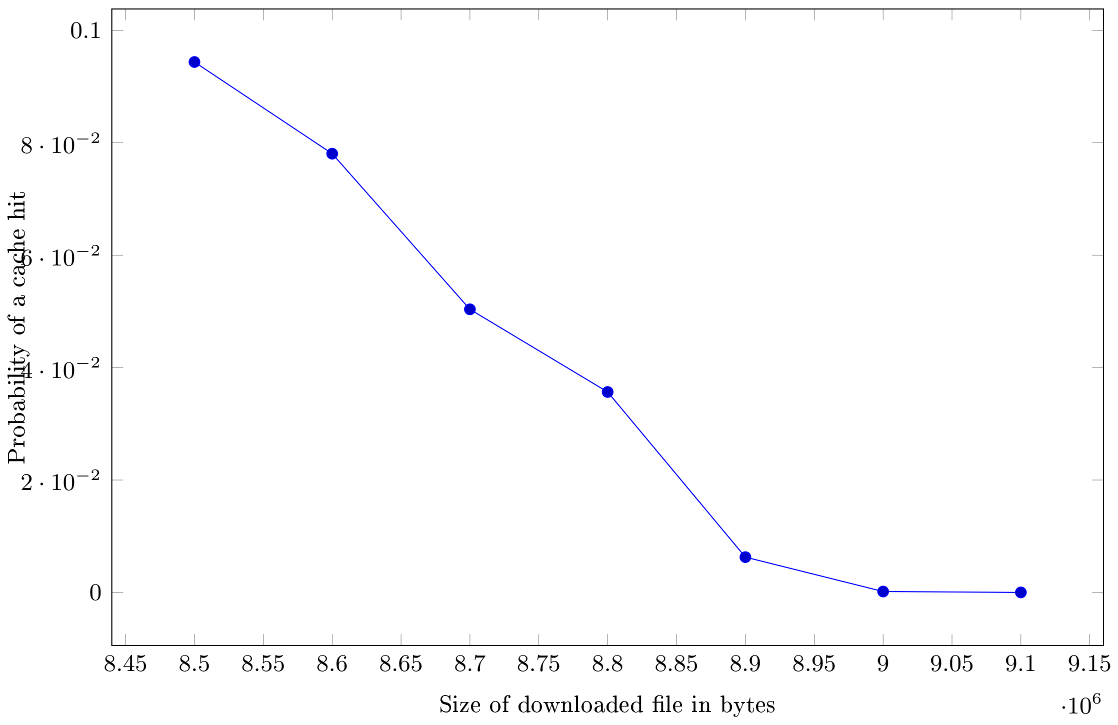


Figure 3.3: Probability of a variable being evicted from the cache simulated via the memset instruction.

3.5 AVX Primitives

We explored a timing difference in AVX instructions which was described by Agner [19]. If a 256-bit instruction gets executed with a value lower than 128-bit, the upper half of the unit is powered down for energy efficiency reasons [19]. If a 256-bit instruction is executed it needs a few nanoseconds to warm up the upper half [19]. In this warm-up phase the unit is capable to perform 256-bit instructions, however it uses 2 128-bit units per 256-bit vector [19]. Using this information, we evaluated, that in a cooled down (passive) state, a single instruction needs more CPU cycles to be performed. Due to energy saving issues a cool down occurs [34].

In our proof of concept implementation, we used the **_mm256_and_si256** (VPAND) instruction. This instruction computes the bitwise logical AND operation between the two variables. Figure 3.4 shows the timing difference between a warmed up unit and a cooled down unit using the instruction. The difference lies on the evaluated CPUs on average between 350 to 450 cycles depending on the CPU state.

This behaviour has an impact on the execution speed of the instructions. Figure 3.4 shows the timing difference between a warmed up unit and a cold unit of i7-8550U. The plot on an i7-6700K desktop CPU looks similar. The instruction on a warmed up unit needs about 600 cycles. A cold unit needs around 1000 cycles. This timing delta is higher than between a cache hit and miss. Thus for a remote attack, it is more robust to noisy channels. We can mount the cache-based attack to AVX. The `leak gadget` is quite similar to the cache leak gadget. Instead of accessing a flag variable which will be cached, an arbitrary AVX2 instruction will be performed to encode a one-bit. This conditional branch can be again mistrained to speculatively execute AVX2 instructions. Listing 3.2 describes how an AVX leak gadget looks like. The `transmit gadget` is an arbitrary AVX2 instruction that will be executed via a request. The mistraining will warm-up the unit. With the transmit, the timing difference can be measured and with sufficient requests, a one-bit and zero-bit can be distinguished.

```
  if (index < len)
    if(bitstream[len])
        _mm256_instruction();
```

Listing 3.2: AVX leak gadget: Conditional branch that performs out-of-bounds accesses via mistraining. The out-of-bounds access of a single bit is encoded by calling an arbitrary AVX2 instruction. [77]

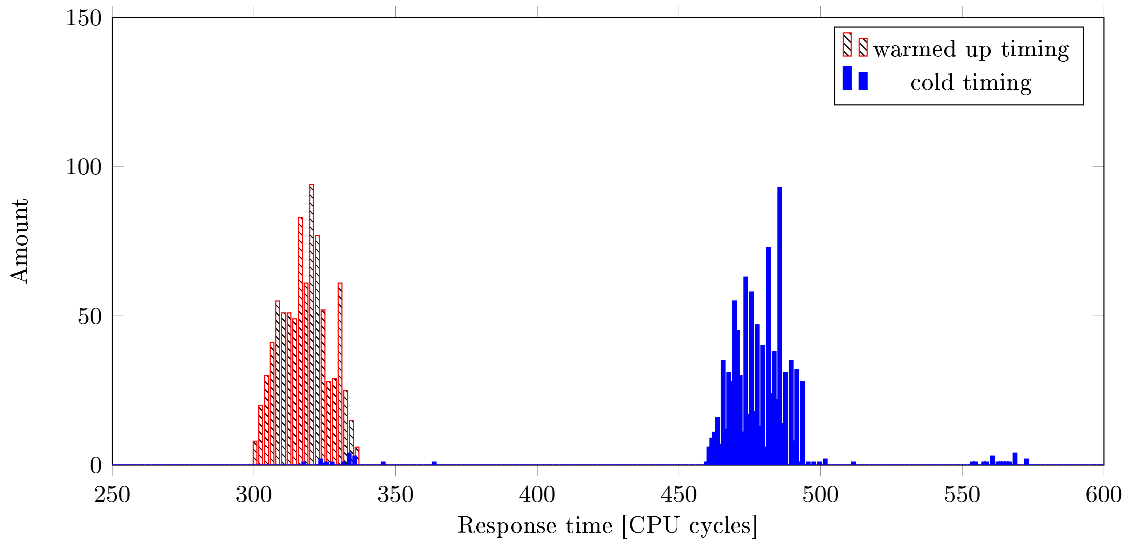


Figure 3.4: Timing difference between a “cold“ AVX instruction and “warm“ instructions

Listing 3.3 contains the necessary code to measure the timing difference of a warmed up and a cold instruction.

Time measurement

To measure the timing difference between a warmed up and cooled unit over the network, we can again choose a histogram-based approach. We first measure the timing of a “cold” AVX instruction and then again of a warmed up instruction. Figure 3.5 illustrates the histogram of the timing difference between a single warmed up and cooled down AVX instruction. It can be seen that a warmed up unit is clearly distinguishable from a cooled down unit. Again, choosing an appropriate threshold is enough to distinguish between a zero and one bit. For our covert channel, we encode a warmed up unit as one bit and the cold unit as zero bit.

```
#define AVX_DIFF 280
#include <immintrin.h>
#include <unistd.h>
...
__m256i a,b;
size_t avx_measuring() {
    size_t start = rdtsc();
    b = _mm256_and_si256(a, b);
    size_t end = rdtsc();

    size_t start2 = rdtsc();
    b = _mm256_and_si256(a, b);
    size_t end2 = rdtsc();

    size_t delta = (end2-start2) - (end-start);
    return delta;
}
```

Listing 3.3: Snippet of AVX timing measurement.

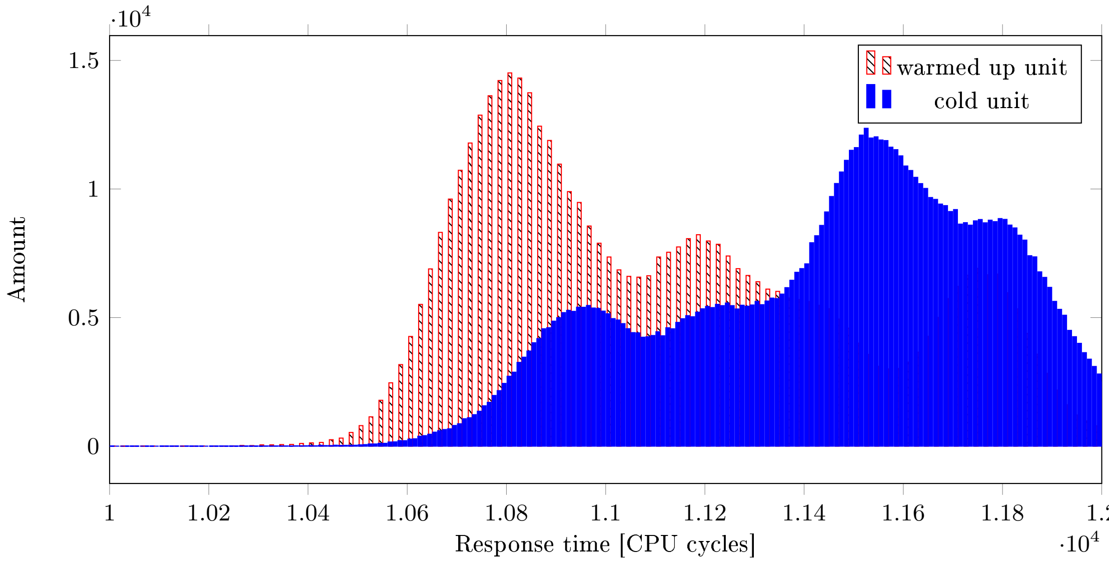


Figure 3.5: The timing difference between a warmed up and cooled down AVX instruction over a network-based measuring.

Resetting of the unit

In order to reset the testing conditions for a single bit, we need to be capable of cooling down the unit. The fastest way to achieve this, is to simply wait a certain amount of time. We observed that after 1 millisecond the unit is completely cooled down [77]. Figure 3.6 shows the cool down of the AVX unit after a certain amount of cycles. After approximately 0.5 milliseconds the upper half of the instruction cools down [19, 77]. The execution time of the instruction increases after increasing the delay. We used NOP (No operation) instructions to increase the delay. From the NOP instructions we can calculate back the time in milliseconds. The reason, why we used NOP instructions was that the **usleep** syscall was too inaccurate on our evaluated CPUs. The zig-zagging effect after 200 and 400 NOPs comes from the scheduling process within the CPU.

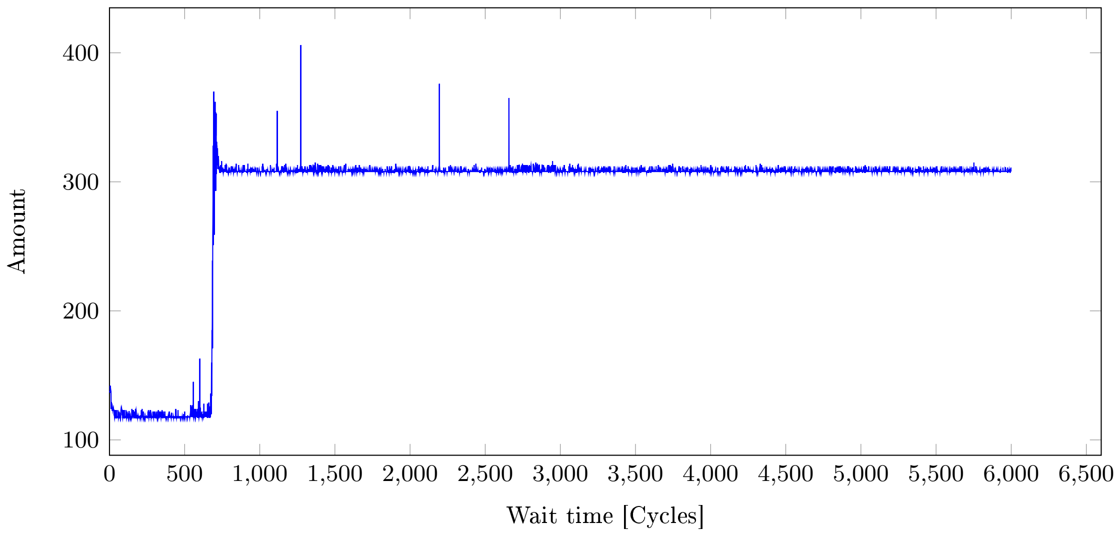


Figure 3.6: Number of NOP instructions necessary to cool down AVX unit after performing the _mm256_and_si256 instruction. After approximately one millisecond the unit is completely cooled down. At approximately half a millisecond the unit starts to cool down.

Chapter 4

Attack Case Studies

In the previous chapter, we defined the necessary attacking primitives to perform a NetSpectre attack. In this chapter, we discuss the implementation of the attack and its performance. Additionally, we evaluate its impact on real networks and the practicality of being attacked over the network.

In the first section, we describe the 2 attack variants which can be used to leak memory from a server application. We demonstrate our attack on current Intel CPUs. The first attack uses a cache covert channel which is based on the idea of Evict+Reload. Instead of flushing, we simulate a file download to reset the cache state like explained in the previous chapter.

In the second variant, we adapt the attack to work with AVX instructions instead of the cache. We achieve a higher performance with this attack since we only have to wait one millisecond to reset the state.

Then, we explain how we can use a NetSpectre gadget to bypass Address-Space-Layout-Randomization (ASLR) on modern operating systems. We use binary search to hit a particular address and accelerate some specific functions. With this approach, we are capable of calculating an offset and break ASLR.

4.1 Leaking Memory

Since the attack is network-based, the primary requirement is a publicly accessible service, which is available via an IP address and a port. This service should be capable of receiving a large number of network packets and have fast response times. We adapted the UDP-socket-based implementations for creating the network histograms. In our implementation, we consider data to be in binary format. This representation makes it easier since we only need to distinguish between a zero and one. Distinguishing a whole byte with 256 possibilities would require a lot more repetitions and a more sophisticated approach to separate bytes. The server holds a binary stream which is divided into a public and private (i.e. secret part). The client implementation can send indices to select bits from the public part. The index reprensents the n^{th} bit of the binary stream. In our proof-of-concept implementation, the server provides the two gadgets:

1. *Leak gadget*: This gadget takes an index to select from the binary stream as input. This input is sent to a conditional branch. The conditional branch checks whether the index is allowed to access the public part or not. If the index is below the allowed public part, the indexed bit is accessed if the bit's value is a one. The conditional branch is vulnerable to Spectre V1 and therefore it is possible to mistrain this branch. The speculated instruction is in our two attack cases as follows. For the cache-based attack, memory is accessed speculatively. In our AVX-based attack, an arbitrary AVX instruction is speculatively executed.
2. *Transmit gadget*: This gadget takes in the cache-based approach an arbitrary index and performs another memory access on the specified data. In the AVX case an additional AVX2 instruction is performed. Depending on a cache hit or a warmed up AVX unit, the response should be below a pre-defined threshold.

To leak a single bit, the attacker needs to use the two NetSpectre gadgets previously defined in two phases.

In the first phase, the attacker continuously mistrains the server's branch predictor via the leak gadget to perform a single instruction speculatively. This single instruction either executes a memory access for the cache-based attack or an arbitrary AVX instruction. The attacker sends valid indices to make the branch prediction believe that the next indices are also valid and trick it to execute the following executions of this branch speculatively. Afterwards, the attacker sends an invalid (out of bounds) index a few times, which then speculatively executes a memory access instruction or an AVX instruction.

After mistraining and accessing the out-of-bounds value of the secret, the attacker triggers the transmit gadget. Here, the memory access instruction or AVX instruction gets again executed. The attacker then measures the response time of the server and compares it to a particular threshold value. We choose as threshold for one-bits the mean value from the 1-bit histograms and compare the response times. Since a single response is not enough and might contain noise, we need to rerun the whole procedure above mentioned multiple times. With this procedure, we are capable of leaking the secret data bitwise.

4.1.1 Cache-based Attack

In our implementation, we aligned the binary stream to a page size of 4096 KB. The server runs three main threads representing the necessary gadgets. The first thread contains a conditional branch containing an index check which is our leak gadget. Our transmit gadget is located in a second thread which always accesses the send index in the array. In the third thread, the reset function is implemented which simulates a file download and flushes the accessed bits. After requesting this thread, the state is completely reset and the experiment can be repeated.

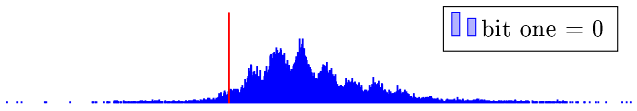
The gadgets look like given in Listing 4.1 beginning with line 11. A memory region is allocated represented as the *mem* variable. If the leak gadget is mistrained, the memory at the location of the indexed bit gets accessed.

A single bit leakage looks as follows. At first, the attacker has to find the threshold using the histogram tool. In the mistraining phase, the client sends for example the number 2. 2, in this case means, the second bit of the bitstream (the array starts with the letter “d” in binary 0110 0100), which is a 1. This index will cause that the branch prediction is mistrained via the leak gadget. If the attacker now sends, for instance, the number 33, the speculative execution will access the second bit of the letter “S”. Using the transmit gadget, the attacker is now capable of testing whether the bit number 33 represents a zero or a one. After the attacker has accessed this bit, he has to reset the cache state using the flushing thread. As stated in the comment on line 13 and 14, a flushing of the index variables x, bit and data will have a higher probability that the branch is predicted and executed speculatively. We observed that with a too small data array, the attack does not work as intended, as there was no speculative execution on the branch. In our proof-of-concept implementation, we used a size of 2048 bytes for the data array, which was large enough, so that the branch was executed speculatively based on the branch prediction.

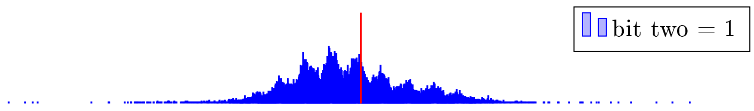
The client repeats this procedure for each bit multiple times. After a certain number of repetitions, the client compares the number of cache hits and misses. Figure 4.1 illustrates the leaked binary encoded letter ‘S’ of the secret part. We choose a threshold of **15500** CPU cycles to distinguish between a zero and one byte. The threshold is marked using a red line. In our proof-of-concept implementation, the plots are not always that distinguishable. Instead of just looking at the histograms and trying to detect zero and one, we count the number of times below the threshold. When using this approach, we observed a significantly higher number of elements below the threshold for one bit than for a zero bit.

```c
unsigned char* data = "dataSECRET..."; // should be large
    enough ~ 2048 bytes
char* mem;

void init_mem()
{
  char* _mem = malloc(4096*300);
  mem = (char*)(((size_t)_mem & ~0xfff) + 0x1000*2 +
     1024);
  memset(mem, 0, 4096 * 290);
}

char leak_gadget(int x,int bit)
{
  //if variable gets flushed beforehand speculation
  //is more likely
  flush(&x);
  flush(&bit);
  flush(data);

  int index = x * 8 * 4096 + bit*4096;
  if(x < (strlen(data)) - strlen("SECRET"))
  {
    return mem[index];
  }
}

void transmit_gadget(int received_byte,int received_bit)
{
  maccess(mem + (received_byte*8*4096 + received_bit *
     4096));
}
```

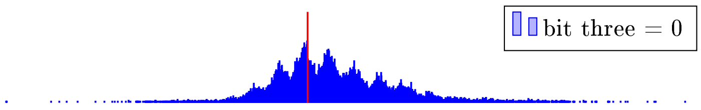
Listing 4.1: Leak gagdet and transmit gadget for cache-based attack.



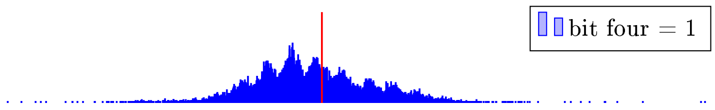
(a) First bit leaked as zero.



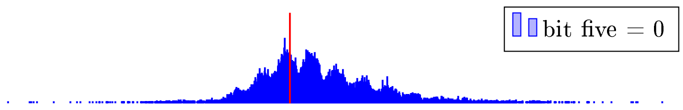
(b) Second bit leaked as one.



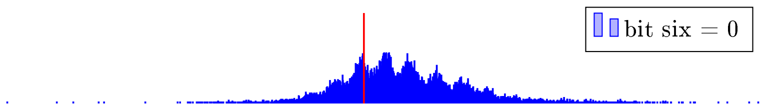
(c) Third bit leaked as zero.



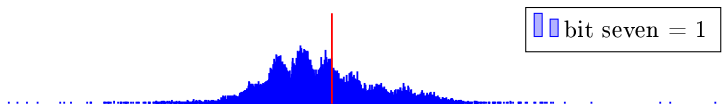
(d) Fourth bit leaked as one.



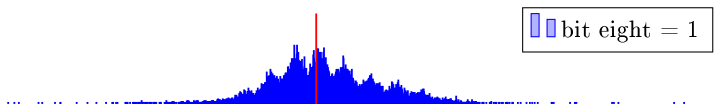
(e) Fifth bit leaked as zero.



(f) Sixth bit leaked as zero.



(g) Seventh bit leaked as one.



(h) Eight bit leaked as one.

Figure 4.1: NetSpectre leakage of the letter 'S' which is 01010011 in binary. A threshold of 15500 cycles is used to distinguish zero and one bits. With fewer iterations, the threshold is used as an indicator. Thus, counting the number of values below the threshold is enough to distinguish the bits. The number of response times below the threshold is significantly higher for one bits, than for zero bits.

4.1.2 AVX-based Attack

To build an AVX-based attack, we need again a leak gadget which can be mistrained. Instead of mistraining a memory access, an arbitrary AVX instruction needs to be speculatively executed. Listing 4.2 gives the two gadgets needed for the AVX-based attack. As can be seen, the condition that gets mistrained is the same as for the cache-based attack. The transmit gadget is a single 256-bit logical AND instruction.

```c
unsigned char* data = "dataSECRET...";
__m256i a,b;
char leak_gadget(int index)
{
  received_byte = index / 8;
  received_bit = index % 8;
  volatile int bit = (data[received_byte] >> received_bit
     ) & 1;
  flush(&received_byte);
  flush(data);

  if(received_byte < strlen(data) - strlen("SECRET"))
  {
    if(bit)
    {
      b = _mm256_and_si256(a, b);
    }
  }
}
...
void evaluation_gadget()
{
  b = _mm256_and_si256(a, b);
}
```

Listing 4.2: Leak gagdet and transmit gadget for AVX-based attack.

The AVX-based attack works in the following way. The server application has two threads. Like in the cache-based approach one thread for the Spectre part of the attack and another one for the evaluation part. The third thread is not needed since we do not need to flush any memory.

The client mistrains the conditional branch, which now speculatively executes an arbitrary AVX instruction. After mistraining, the client sends one single request to the leak gadget, which again executes an AVX instruction and measures the response time. For a one bit the instruction should be again faster on average than for a zero bit.

To reset the attack per bit the client now only has to wait 1 millisecond to cool down the AVX unit. Afterwards, the attack procedure can be repeated like before.

4.2 Bypassing ASLR

For remote attackers, ASLR is often one of the last hurdle which has to be beaten.

The idea of an ASLR bypass using NetSpectre is to speculatively access an arbitrary address used in the process. Due to a cache hit the execution speed of this function should be faster. In this case, we only need one gadget, where we are able to mistrain the branch prediction to speculate over this function. An ASLR gadget leaks information about a memory access [77]. With this gadget, the attacker is capable to speculatively access any memory address. Listing 4.3 illustrates an ASLR gadget.

```
if (index < len)
  access(array[index])
```

Listing 4.3: NetSpectre ASLR gadget [77]

There exists a page without randomisation called the **vsyscall** page [32]. For each process, the vsyscall is located at the same memory address. This page is used to speed up specific system calls. The three functions *getcpu, time, gettimeofday* are provided in the vsyscall page. If the KAISER patch [24], which prevents Meltdown, is enabled, the vsyscall is the only page mapped in a certain region.

The ASLR leakage with the leak gadget and the vsyscall page works as follows. In the first phase, the attacker mistrains the branch prediction using the ASLR gadget. The aim is to hit the vsyscall page which then accelerates one of the mentioned functions above. The server needs to provide a public interface which accesses one the three above mentioned functions. The attacker measures the response time of the interface. If the response time is faster, a cache hit got detected.

The attacker can perform a binary search with all offsets to reach a certain memory region e.g. the vsyscall page. The number of offsets gets bisected. First the attacker mistrains the branch predictor and uses the ASLR gadget with the first half from the offsets to speculatively access all virtual addresses in the range. Afterwards, the public interface is called and its response time is measured. If the response time appears to be faster, we choose the first half and bisect this half again. If not, the other half of memory is chosen and the check is repeated. This type of search brings a logarithmic search depth. Meaning for an ASLR entropy of around thirty bits, we only need thirty checks to find the correct offsets.

Figure 4.2 illustrates the binary search performed to find the correct offset. The first node is the first half of memory in which a single hit occurs. The vsyscall page is located at ffffffffff600000.

The vsyscall page is actually in use at the latest versions of Linux systems like Debian, though it is a legacy concept [77]. The three functions provided in vsyscall are seemingly harmless, although they give at least a simple ROP gadget called *syscall and return gadget*. Those gadgets can be used in advanced Return-Oriented-Programming techniques like Sigreturn-Oriented Programming [7]. It was replaced by a new concept called vDSO(virtual Dynamic Shared Object) [32]. VDSO is in contrast to vsyscall dynamic and maps memory randomized for each process into a virtual shared object. Using this concept, it is not possible to use this fixed page to bypass ASLR.

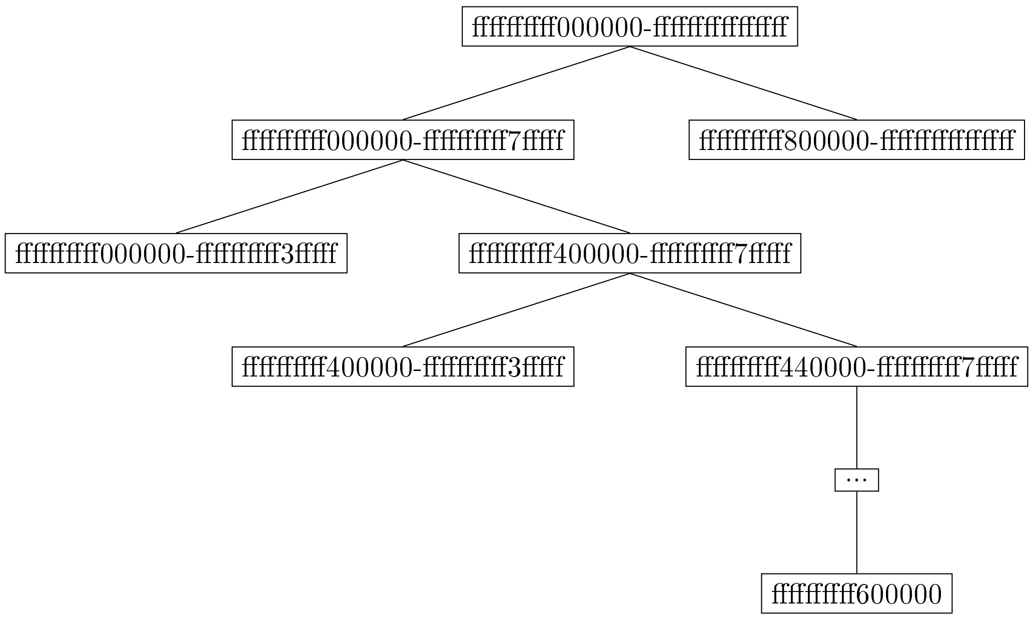

ffffffffff000000-ffffffffffffffff
ffffffffff000000-ffffffffff7fffff
ffffffffff800000-ffffffffffffffff
ffffffffff000000-ffffffffff3fffff
ffffffffff400000-ffffffffff7fffff
ffffffffff400000-ffffffffff3fffff
ffffffffff440000-ffffffffff7fffff
...
ffffffffff600000

Figure 4.2: Illustration of binary search used to find the correct offset.

Chapter 5

Evaluation

In this chapter, we discuss the results of the proof-of-concept implementation.

First, we discuss why we chose the UDP socket connection between the client and server.

Secondly, we evaluate the performance of all three attacks on current Intel CPUs in the local network, the Google Cloud Platform and on an ARM CPU. We argue which other Spectre variants could be used in a NetSpectre scenario. We discuss NetSpectre mitigations over the networking side. Additionally, we analyse current Spectre remediations.

Finally, we discuss the general impact and practicability of NetSpectre. We argue why other attacks which had first a low performance, became later practical.

5.1 UDP Protocol

The UDP protocol has less communication overhead compared to TCP [42]. Therefore using UDP, the performance is higher. However, it is also unreliable and it occurs that packets get lost. Since the measurements are repeated a lot of times, the unreliability in the form of corrupt packets does not matter significantly. We observed that UDP socket timings also vary depending on the CPU workload. Additionally, at the beginning of the first few requests, the timings appear to be a little faster and can be ignored.

5.2 Test Results

We evaluated our implementation by leaking multiple bits on the localhost, a peer-to-peer connection between two PCs and on a cloud environment. The performance is indicated by the true capacity of the channel. Our evaluation CPUs are several Intel notebook and desktop CPUs. Additionally, to desktop and notebook CPUs, we evaluated NetSpectre on the Google Cloud platform and also on an ARM mobile CPU. Table 5.1 lists all evaluated CPUs. We used our proof-of-concept implementations described above in a local environment. As already mentioned, these implementations all use Spectre variant 1.

Vendor	CPU name	Type
Intel	i5-4200M	Notebook CPU
Intel	i5-6200U	Notebook CPU
Intel	i7-8550U	Notebook CPU
Intel	i7-6700K	Desktop CPU
Intel	i7-8700K	Desktop CPU
Intel	Xeon (unspecified)	Google Cloud CPU
ARM	Cortex A75	Mobile CPU

Table 5.1: All evaluated test CPUs for the NetSpectre evaluation. The CPUs evaluated are state-of-the-art CPus.

Intel notebook and desktop CPUs

The bottleneck of the cache-based attack is the cache eviction part, which we described as a constant file download from the server part. It slows down the server and needs to be performed constantly, to flush the accessed bits again out of the memory. In the cache-based approach, we can test for multiple cache hits and misses in a shorter time or parallelized. However, we attain only a transmission rate of 240 bits per hour in a local network with an error rate of less than 0.1%. In a local network, around 100.000 repetitions are necessary to distinguish a zero from a one bit. The more repetitions, the higher the probability to clearly distinguish the bits. To reliably leak a single bit on a peer-to-peer 1 Gigabit link, we observed that at least 1 000 000 repetitions are necessary. For 1 000 000 repetitions, we need around 4 minutes per bit. Thus, the cache covert channel on a local network has a channel rate of 15 bits per hour.

In the AVX-based approach, it is not possible to test multiple bits at once, since we can only warm up and cool down the unit. However, there is no need to evict/flush variables out of the cache. Instead, we only need to wait 1 ms to cool down the unit on a desktop CPU. The throughput is still higher than on the cache-based approach. We need about 1 minute to leak one bit. Using the AVX-based covert channel, we are capable of transmiting 1000 bits per hour in a local network using 100 000 requests. The corresponding error rate is at 0.58%. In a peer-to-peer network, we leak 60 bits per hour.

On Intel i7-8550U, a notebook CPU, a timing difference of up to 1000 cycles was observed, between a cooled down and a warmed up unit. This is a big advantage since the larger timing difference allows more noise on the network. Therefore, it is easier to distinguish between a warmed up and cold unit than between a cache hit or cache miss. On desktop CPUs, the unit is immediately warmed up after a single instruction. On a notebook CPU, for instance the Intel i7-8550U, around 20 AVX instructions are necessary to warm up the unit.

The ASLR bypass is required to use the cache-based attack. Thus, to break ASLR on average an attacker needs to leak 30 bits which takes approximately 2 hours.

5.2.1 ARM architecture

We evaluated our attack also on an ARM Cortex A75 CPU on a local network. A wireless network would be more realistic for a mobile CPU. However, this would bring too much noise to our attack [77]. On ARM, the timing gap between a cache hit and a cache miss is up to 400 CPU cycles [53].

Figure 5.1 shows the histogram of a cache hit and a cache miss on an ARM Cortex A75. As can be seen, we again have a clear and distinguishable timing difference. On the local network again around 1 million repetitions are necessary to leak one single bit. Using a proper threshold, a cache-based attack on ARM is also possible. However, the latency on the CPU is a few times higher than on desktop CPUs. This fact slows down the local attack and the number of leaked bits per hour.

We evaluated the pendant of AVX instructions on ARM. Those instructions are called ARM Neon instructions [4]. However, we did not observe a warmup and cooldown behaviour with these SIMD instructions.

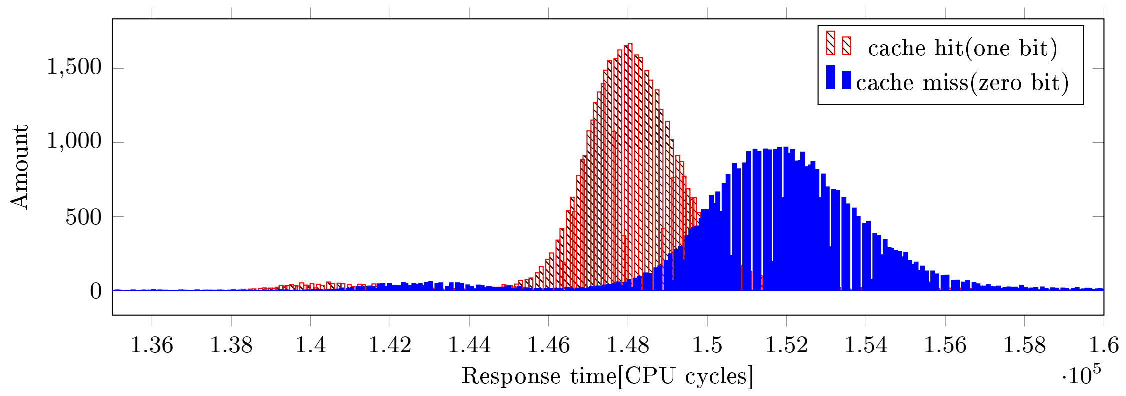


Figure 5.1: Histogram of a cache hit and cache miss on an ARM Cortex A75.

5.2.2 Google cloud

We evaluated NetSpectre on the Google Cloud Platform on two neighbored instances. Those virtualized instances run Ubuntu 16.04.4 LTS [77]. In order to successfully leak a single bit, we need in total 20 million repetitions [77]. The network used in the Google cloud has a bandwidth of 4 Gbit/s. The results are quite noisy, however, using this large number of repetitions a zero or one bit can be distinguished. Thus, in order to leak a single bit, we need about 3 hours in the AVX case and 8 hours in the cache-based approach. Furthermore, we evaluated NetSpectre on two virtualized Xubuntu 16.04 instances using VirtualBox on a local setup. The response times are noisy but with 5 000 000 repetitions, those are still distinguishable.

Test Improvements and Observations

We observed that *stress -d 1* additionally increases the probability for a successful speculation. Furthermore, interrupts, for instance, the movement of the mouse, increase the probability of speculative execution. A possible reason for that can be that the CPU has to speculate more to maintain its performance. However, too much stress on the CPU reduces the success probability of the attack since it produces additional noise. Furthermore, for notebook CPUs, a suspension of the operating system creates an unexplainable behaviour for both cache and AVX-based attack, which we cannot explain so far.

Additionally, we observed that bounding a process to a certain CPU achieved as better performance, in the mistraining phase. To taskset the programs to a single core we used the *taskset -c <core-number> ./process* command. For the AVX-based attack *stress* command adds additional noise and we did not use it to test the attack. Our assumption is that the stress command might have an influence on the AVX unit. As previously mentioned, flushing of the index variable before the conditional branch increases the probability that the conditional branch is speculatively executed.

When testing the cache-based attack locally, the process should be scheduled after mistraining to achieve better results. This can be done by using the **sched_yield**. Reason for that is that the server will always have time to flush the cache state again.

Due to the power scaling of modern laptops, the best results are achieved if the laptop is connected to the power supply. In battery mode, we observed that the Spectre attack is not reliably working anymore. This behaviour might occur due to power saving reasons since the CPU gets throttled down. If experiments are performed in battery mode, the threshold for all attacks has to be calculated dynamically . For notebook CPUs it might occur that one AVX instruction is not enough to warm up the unit. Thus, there is no need to warm it up a little stronger and using Spectre make it "hot" and then measure the timing. Also, it might occur that the timings of AVX shift in a certain way a little around. Therefore it is meaningful to store all the response timings and analyse the data for certain timing sections.

5.3 Porting NetSpectre to other Spectre Variants

In this section, we discuss the portability of NetSpectre to other versions than Spectre V1. We first explain the usages with Spectre V2 up to Spectre V5. We discuss the changes that have to be made, to successfully adapt the attack.

Spectre V1.1

Spectre V1.1 (Bounds check bypass on stores) is similar to V1 with the contrast that the attacker is capable of writing an arbitrary value speculatively. Listing 5.1 shows a conditional branch that can be exploited using Specter V1.1. Here, the two variables *idx* and *val* are controlled by the attacker. With these values, the attacker can trigger speculative buffer overflows. This branch can be directly used as NetSpectre gadget. Here we can speculatively store arbitrary data on the array. For NetSpectre, we can use this gadget to be mistrained and cause arbitrary writes with respect to the data array. By speculatively overwriting the return address, we can trigger a speculative buffer overflow. Combining this attack with the ASLR bypass, we are capable of building speculative ROP chains.

```
if (idx < certain_length)
{
  data[idx] = val;
}
```

Listing 5.1: Spectre V1.1 gadget [47]

Spectre V2

In Spectre V2, the attacker injects an arbitrary target address to the BTB [49]. This overwritten target address will be fetched and the code speculatively executed. To adapt NetSpectre to Spectre V2, we need to have adapted gadgets. Spectre V2 can be used to exploit C++ vtables. The vtable is a pointer to an array of function pointers. Whereas, each offset in this array represents a different virtual function. The Spectre gadget, in this case, can be a virtual function. The NetSpectre V2 scenario could look as follows. The server side contains a super class which has for instance two sub classes. Both classes contain the same function. For one class this function is public and does nothing dangerous. For the other class this function contains a secret that can be leaked via Spectre V2. In the mistraining phase, the attacker needs to mistrain the harmless function to inject the target address into the BTB. Afterwards, the attacker tries to access the secret function which should be speculatively executed. Again, in this speculative execution, a memory access on a bitstream or a SIMD instruction gets executed. The second gadget we need is again the transmit gadget either for cache or AVX. To flush the cache, we again need the possibility of a constant file download. We evaluated Spectre V2 locally and observed, that 5 mistraining requests are enough to execute the secret function at least once.

Spectre V4 (Speculative Store Bypass)

Spectre V4 exploits the fact that in modern processors speculative loads are processed even if the address of an overlapping store operation is not known [8]. Thus, our Spectre gadget has to contain a store instruction and directly afterwards a load instruction. It is more or less a race condition here, and no mistraining can be performed. Mainly this problem can be similarly exploited like variant 1 and 2. Listing 5.2 shows the code that has to be executed on the server side. A binary mask is used to filter out values for the *idx* variable. Due to an error in the store-to-forward logic, the access on the data array with the old value is performed speculatively [49]. However, the attacker has to perform a lot of requests to trigger this race condition.

```
void v4gadget()
{
  int idx;
  int* p_idx=&idx;
  (*p_idx)&=mask;
  data[ptr[idx]*4096];
}
```

Listing 5.2: Spectre V4 gadget

Spectre V5 (RSB)

Spectre V5 aims to exploit Return Stack Buffers via speculative execution [58]. Maisuradze et. al. describes four scenarios to mistrain the RSB [58]. In those, the RSB gets manipulated via *call*, *ret* and *pop* instructions.

There is no direct possibility to manipulate the stack like this using Spectre and it has to be provided in the code section. However, if there is the possibility to control the stack pointer, the attacker would use way more dangerous attacking techniques like Return-Oriented-Programming to get control over the full system.

Meltdown

Meltdown allows an attacker to read arbitrary memory without any necessary privileges [54]. During transient execution the user accessible bit is not checked and allows it to access kernel memory from userspace [54]. Foreshadow was proposed by Van Bulck et al. [83] on SGX. In this attack the present bit gets cleared to ensure that a pagefault occurs. An L1 lookup with the virtual address is performed directly without checking the permissions [83].

Spectre V1.2 allows an attacker to speculatively overwrite read-only memory [47]. Read-only memory can be read-only data, code pointers, and code metadata, including vtables, GOT/IAT, and control-flow mitigation metadata [47].

So far, these attacks are considered to be local. We did not find a possibility porting these attacks to the network. Since signals have to be handled which is quite uncommon for server-side applications.

We consider this as an open research problem, which needs some further investigation. Furthermore, the patches against Meltdown are effective and widespread.

5.4 NetSpectre Countermeasures

In this section, we explain how to mitigate NetSpectre via the network side. We explain how current networking devices are capable of detecting networking anomalies and how fast they get blocked. Furthermore, we discuss current approaches to remedy Spectre and argue why these countermeasures do not fully prevent NetSpectre.

Networking countermeasures

The first possibility to mitigate NetSpectre is via network traffic monitoring. Next-generation firewalls, routers but also more sophisticated intrusion detection systems are capable of inspecting, analysing and detecting networking anomalies [6, 79, 82, 93]. If the sent payloads look malicious, they might get blocked by networking devices quite fast. For instance, in the cache-based attack, performing a constant file download on the server is definitely suspicious enough to get blocked. Furthermore, the number of packages which are necessary to leak one bit is quite high and might be detected as Denial-Of-Service attack attempt. A network firewall or router can be easily configured to limit the traffic number per source IP address [6]. Thus, it is easy to detect such attempts and block them [79]. Furthermore, random delays or load balancing respectively normal network traffic delay weaken the attack a lot. The number of devices between the victim and the attacker also adds a lot of noise. A proper network segmentation or demilitarized zones for sensitive services can also be considered to harden NetSpectre attacks. Additionally, IP address whitelisting can be used in internal networks for remotely accessible services like RDP, SSH or LDAP.

However, if the attacker has time to wait, the networking packets could be obfuscated and sent in different slow intervals. Furthermore, fragmenting payloads is an opportunity to evade intrusion detection systems [70]. The detection system has to learn the signature of certain payloads. Since the way of using NetSpectre can be different for each case, there is no guarantee that these requests are blocked.

Side-channel countermeasures

In order to overcome NetSpectre attacks, we need to patch Spectre in all its variants. Since the vulnerability is located in the hardware the simplest but most expensive mitigation is to disable branch prediction and with that speculative execution [9, 49]. One could also switch to processors that do not support speculative execution for instance some ARM processors [49]. Alternatively, one can wait and change to CPUs that are not affected by Spectre. This would also have a huge impact on the performance [49]. So far, there is also no possibility to turn-off speculative execution completely using oftware [49].

In the following subsections, we will discuss the possibilities to mitigate Spectre and other side-channel attacks.

Hardware-based defense

Khasawneh et al. [45] developed an approach which uses shadow hardware structures, which are especially used for transient instructions. These structures check whether the predictios were correct or not at runtime. If the predictions were not correct, the results are discarded and no side-effects in the hardware occur.

Yan et al. [90] proposed a method which is called InvisiSpec. This method uses a speculative buffer, which stores all speculative executed loads instead of using the cache. If the speculation was correct the results are loaded into the cache. If the speculation was wrong the results get invalidated [90]. This approach only protects the cache so far, and not other covert channels like the AVX-based approach.

Another solution named DAWG was proposed to secure the cache by protection domains which perform cache isolation [21]. However, this approach requires to redesign the cache and requires a correct handling of these protection domains [8].

Add noise to times

One possibility to mitigate side-channel attacks is to add noise to the timers [8]. For client-side attacks for instance performed on the browser this could be considered as remediation. Since the accuracy has to be precise, noise on the timers makes it harder to detect changes in the microarchitectural state. This idea was implemented in Chrome and Edge browser to mitigate Javascript side-channel attacks [29].

However this solution is not ideally, since there are other possibilities to create timers like Schwarz et al. [76], showed.

Memory barriers

As already mentioned, the microcode updates from Intel only mitigated the possibility to leak kernel memory from a userspace application [36]. The first possibility recommended by Intel was to use memory barriers. Intel proposed to use memory barriers in the form of the *lfence* instruction to mitigate the first variant of Spectre exploiting conditional branches [36]. This instruction prevents later instructions from being speculatively executed. Listing 5.3 shows how the memory access in the Spectre gadget could be prevented in the conditional branch.

We evaluated the impact of using a single lfence before a memory access for the cache-based attack and the AVX-based attack. For the cache-based attack, a single lfence is

```
if(x < (strlen(data)) - strlen("SECRET"))
{
  asm volatile("lfence");
  return mem[index];
}
```

Listing 5.3: lfence instruction used to fix Spectre

enough to mitigate the attack. However, for AVX we discovered that a single lfence instruction `does not` prevent NetSpectre attack. Figure 5.2 illustrates the timing difference of a cold and warmed up unit with an additional speculatively executed AVX instruction. In the leak case, without an AVX instruction, the timing difference is most significant. However, with a single lfence instruction, the timing difference between a cold and warmed up unit is still distinguishable. It can be seen, that in the baseline case, where only a single lfence gets executed without an additional AVX instruction, the unit is not warmed up. For the ASLR attack, we recommend to turn off the vsyscall page by using the *echo 0 > /proc/sys/kernel/vsyscall64* command.

Adding lfences for each critical section can be a hard task. For productive systems, it is sometimes hard to post patch the system without losing uptime. For high-performance applications, each lfence reduces the speed of the application. Furthermore, it is easy to overlook or forget a critical section when applying lfences.

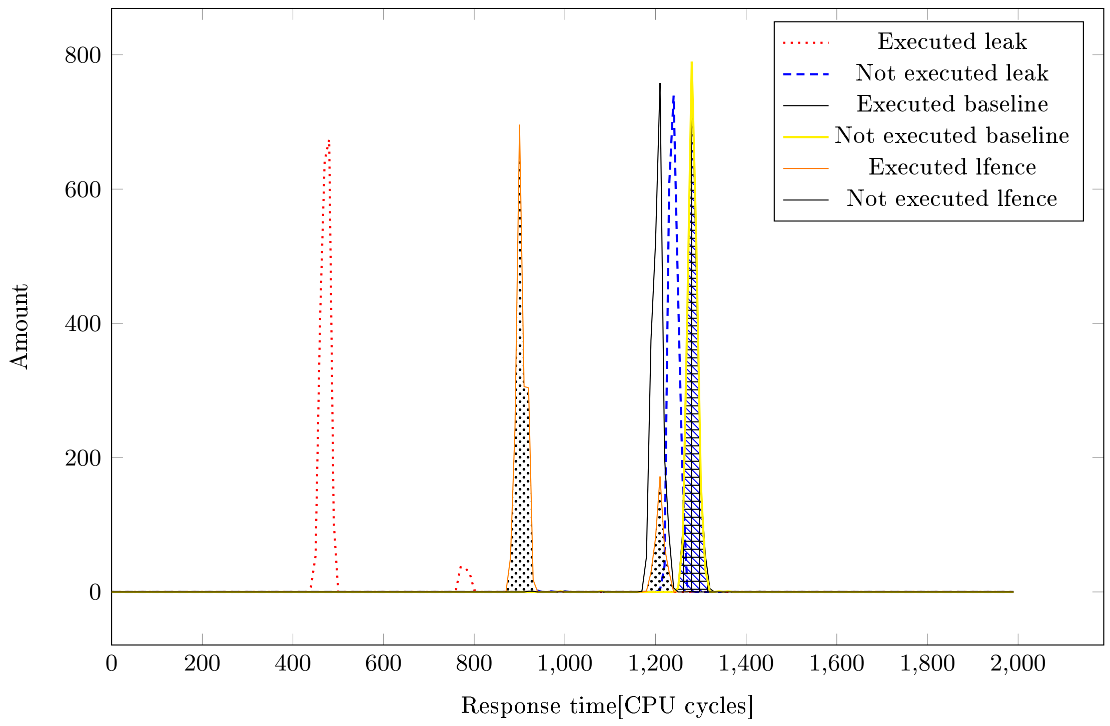


Figure 5.2: Timing difference of a single AVX instruction sped up speculatively. It can be seen that adding a lfence does not completely prevent an AVX instruction to get executed. The warmed up unit is still distinguishable from a cold unit.

Compiler-supported gadget detection and patches

The detection of Spectre gadgets is important to prevent Spectre attacks. Thus, we recommend to automatically detect and patch potential NetSpectre gadgets using a proper tool like Coccinelle used for patching bugs in the Linux kernel [73]. Red Hat Enterprise [71] developed a tool that emulates the execution of the binary and tries to follow each path of conditional branches. The tool inspects register values and certain memory regions like stack, heap and the data sections. If the tool detects a potentially vulnerable branch, the address in the data section is reported. The number of false positives might be relatively high using this approach. However, static analysis could fail therefore a manual inspection of the code is always recommended.

Wang et al. [86] proposed a solution for Spectre V1 gadget detection and patching via binary analysis. At first they extract the control flow [86]. Then they use taint and address analysis to detect potential vulnerable conditional branches and are capable of patching vulnerable code section.

The Microsoft Visual C compiler provides a compiler supported feature that is capable of mitigating Spectre V1,V2 and V3. Their approach is to detect and patch vulnerable code patterns using static code analysis [49]. Their V1 mitigation also uses lfence instruction, which we showed are no fix for the AVX-based attack. As Kocher stated [48], the implementation is very likely to miss vulnerable code snippets. A feature to mark data secrets as uncacheable can be also be considered to mitigate cache attacks [8]. For large secrets in cryptographic operations, this might have a significant performance impact.

Retpoline and other Spectre variant mitigations

In this thesis, we did not completely evaluate Spectre V2. However, we discussed ways to exploit Spectre V2 using NetSpectre [77]. Google Project Zero proposed [94] to mitigate Spectre V2. This mitigation technique uses a "return trampoline", which runs in an infinite loop. This loop causes the CPU to not speculate over the indirect branch. Retpoline can be integrated to compilers like the Gnu C Compiler and LLVM [56, 57].

Meltdown is mitigated using a technique called KAISER [24]. This technique separates page tables for userspace and kernelspace.

5.5 Discussion

Although NetSpectre works well only on the local network, this attack could have serious consequences. Considering targeted attacks, where we only need to leak a certain secret with fewer bytes, the attack can be quite effective. Often a small piece of information is enough to gain further access to a system. It is easy to create and plant NetSpectre gadgets into the software. Besides, it is also quite simple to create malware that attacks targets via NetSpectre. For instance a malware, placed in a large company could attack using NetSpectre for months without getting detected, if the attack is not too conspicuous [82]. Leaking bit by bit until keys or user credentials are leaked. Using this sensitive data, further attacks in the network can be started. So far, we haven't found NetSpectre gadgets in real-world applications. NetSpectre gadgets are not easy to detect but they might still be located in web server implementations, network drivers or other network related implementations.

Many similar attacks were first shown in theory or in a laboratory environment. For instance, Rowhammer [46] which flips bits in the RAM was at first also a more or less theoretical attack. In 2018, van der Veen et.al [84] presented RAMpage and its mitigation Guardion. With this attack, they are capable of rooting Android mobile phones via bitflipping. Another attack which first shown theoretically is Lucky Thirteen [1] which then broke the TLS protocol. However, this also needs a large number of sessions, to be capable to decrypt the content. Nevertheless, with enough resources, the attack is feasible.

The potential of these types of attacks should not be underestimated. With faster link connections and lower latency times, the speed of these attacks increases proportionally. Furthermore, investing more time to optimize the attack could increase the performance substantially. When the moment comes where these attacks achieve a good performance and become attractive for malicious activities, the countermeasures should be strong enough and already deployed.

5.5.1 Limitations

The results of this work seem to be quite slow, and in fact, this type of attack can be easily mitigated as discussed in the countermeasures section. Furthermore, the measurements were taken in a lab environment. The representation in binary is unusual for common applications. However, we showed, that our experiments can also be reproduced in productive environments like Google Cloud instances. Of course, the direct impact is as far not as big, compared to other vulnerabilities like Heartbleed [13]. As we discussed in the previous section, Spectre V1 will probably not be completely fixed and does not have a performant fix at the moment. With approximately more than one billion vulnerable devices, this remote attack has indeed an impact.

Chapter 6

Future work

In this chapter, we outline future research tasks that can be performed and what problems remain. Since Spectre attacks are quite recent, there exists much space for more sophisticated and more efficient attacks.

The most difficult challenge is to detect and mitigate Spectre gadgets. Since there probably will not be a complete fix, it is crucial to secure the state-of-the-art CPUs with proper countermeasures. As we discussed in the previous chapters, this is a quite complicated task. There exists a big gap between performance and security. In the past decades, hardware vendors tried to make their chips as fast as possible. However, security was not considered enough in the design. Their task is to redesign their chips without losing the current performance. The currently proposed mitigations like memory barriers and Retpoline should be evaluated for potential unwanted side effects [77]. It is also not completely validated if these tools, do not coincide with other security mechanisms. We at least defined the NetSpectre gadgets in a general form. However, there might be code fragments which might lead to a similar behaviour like Spectre gadgets. Static analysis is probably not enough to detect all Spectre gadgets. Maybe there are ways to efficiently detect the mistraining part of NetSpectre on the software side.

One research task could also be to evaluate NetSpectre attacks on current state-of-the-art network security devices. Maybe there is a way to bypass those security devices and improve their detection rate. We have not evaluated the results on AMD CPUs. Thus, evaluating and testing NetSpectre on AMD CPUs could be done. Furthermore, an evaluation of wireless networks would be interesting. Maybe there is a way to reduce the noise. The AVX attack could be ported and evaluated on other attacks like AMD with the use of AVX instructions. We described how to port the attacks to other Spectre variants theoretically. Additionally to the Spectre variants, the way of attacking can be adapted to other side-channel attacks. Those variants could be implemented and evaluated. Furthermore, there is for sure more ongoing research on Spectre. Attacks like Branchscope [17] and Foreshadow [83, 88] might also be adapted to work over the network to attack virtual machines and their hosts.

The current performance of NetSpectre can be improved by using more sophisticated approaches. For instance, the thresholding approach could be replaced by a Machine Learning approach. By learning the response times and clustering, it might be possible to reduce the number of repetitions. The part of mistraining seems to be optimal locally, whereas five valid indexes are enough to send at least one out-of-bounds index. Maybe there is also a way to parallelise the AVX based approach to leak more than one bit.

Our proof-of-concept implementation only considers binary data. The attack can be ported to distinguish between bytes. However, this would require a lot more repetitions and a more sophisticated approach to recognise different bytes.

6.1 Detection of Spectre gadgets

As already mentioned the automated detection of is a novel research topic. We created two approaches to statically detect Spectre V1 gadgets and ASLR gadgets in binaries and source code. Our first approach detects Spectre V1 gadgets on C code with the static code analysis tool Coccinelle [73]. In the Coccinelle script, we defined how an ASLR gadget looks like. Listing 6.1 contains the Coccinelle script used to detect potential Spectre V1 gadgets. We tested our script on several open-source projects for example on the Linux kernel, Apache and NGINX web server and OpenSSH Server implementations. For huge code bases, the effort to analyse all branches is quite high, since the false positive rate is quite high.

Since the source code is not always present and NetSpectre gadgets can also be detected in binary, we created a second static approach to detect gadgets for executables. We used the Capstone engine to detect potential Spectre gadgets in binary statically [14]. In this approach, we parsed the binary, iterated over the symbols table and checked for conditional branches with bounds checks, where the index of the branch is attacker-controlled.

The better approach to detect Spectre gadgets is via dynamic analysis like shown in the promising framework by Wang et al. [86]. In this approach, they used taint analysis to detect Spectre gadgets. They define constraints for a Spectre gadget. The constraints are a branch with an out-of-bounds check of external input and a speculative read or write on a data array which produces side-effects on the microarchitectural state. Branches which fulfil those requirements are marked as tainted. The framework also patches tainted branches with *lfence* instructions.

So far we have not found an existing NetSpectre gadget, but we still assume that there are NetSpectre gadgets in the wild.

```
virtual context

@r1 depends on context@
expression v,u ;
position p1, p2;
identifier I, e, X, f;
statement s;
@@

f < v@p1
<...
(
* I[e]@p2
|
* *(I + e)
)
...>
```

Listing 6.1: Coccinelle script to detect Spectre V1.1 gadget

Chapter 7

Summary

In this thesis, we showed the possibility to exploit Spectre variant 1 under a networking aspect. This is a new way to remotely exploit a system using a hardware vulnerability. We defined the necessary code gadgets for our two attacking variants. Via a histogram-based approach we showed that local timing differences in the cache and SIMD instructions are recognisable in response times.

We implemented a proof-of-concept server-client scenario in C. We created a network-based cache attack variant, which allowed us to leak at least 15 bits per hour in the local network. To overcome the problem of a constant flushing, we observed a similar behaviour of cache eviction via a constant file download. We called this approach of attacking Thrash+Reload.

We found a new timing side-channel using Intel AVX instructions, which we additionally exploited in this thesis. This attacking type is even more performant in our proof-of-concept implementation. Using the AVX-based approach, we are capable to leak at least 60 bits per hour in a local network. Using AVX, we created a new way to communicate via a covert channel.

Our experiments got successfully evaluated on current Intel desktop and notebook CPUs, an ARM CPU and an CPU of the Google cloud environment. Furthermore, we showed a way to bypass ASLR using our cache-based attack with a necessary leakage of approximately 30 bits. We discussed the other Spectre variants and how to port the relevant attacks to NetSpectre.

We examined networking and Spectre countermeasures to mitigate NetSpectre. We observed that a single lfence is not enough to remedy the AVX-based attack. Furthermore, we discussed the practical impact of NetSpectre and named comparable examples.

We additionally outlined future work open for research. There exist a lot of new challenges for instance migrating side-channel attacks to network. This makes NetSpectre a realistic threat to millions of devices.

Appendix A

Black Hat Asia 2018

We submitted a paper with the title "NetSpectre: A Truly Remote Spectre Variant"" to Black Hat Asia 2019 which got accepted for a 50-minute talk. Together with my advisor Michael Schwarz, we will give a presentation about this thesis at Black Hat Asia 2019.

A.0.1 Abstract

Modern processors use branch prediction and speculative execution to increase their performance. Since January 2018, with the publication of Spectre attacks, we have seen that speculative execution can be abused to leak confidential information. By inducing a victim to speculatively perform operations that would not occur during correct program execution, confidential information can be leaked via a side channel to the adversary. Many countermeasures and workarounds have been proposed, all assuming that Spectre attacks are local attacks, requiring an adversary to execute code on the victim machine.

In this talk, we present NetSpectre attacks. We show that Spectre attacks are not limited to local code execution but can even be mounted remotely over the network. NetSpectre attacks can be mounted without any user interaction, just by exploiting Spectre-like gadgets exposed to the network. We show that such an attack is not only theoretically possible by presenting data leakage across virtual machines in the Google cloud.

We will then discuss why Spectre mitigations are incomplete and do not prevent Net-Spectre. By demonstrating a novel variation of Spectre, which uses a previously unknown side channel, we show that the assumptions of many countermeasures are wrong, making these countermeasures ineffective. Thus, we emphasize the need for more research on such attacks to find better countermeasures.

We outline challenges for future research on Spectre attacks and mitigations. Finally, we will discuss the short-term and long-term implications of Spectre as well as NetSpectre

for hardware vendors, software vendors, and users.

A.0.2 Presentation outline

1. **Introduction to Speculative Execution**
   We start with a simple example of branch prediction. We start with a code construction that is commonly found in sandboxed and non-sandboxed code, a bounds check. Passing legitimate values to the bounds check will train the branch predictor to take the then-branch. Subsequent calls with out-of-bounds accesses, however, also lead to speculative execution of the then-branch. We will see how microarchitectural covert channels can leak information from the speculative execution to a persistent state.
2. **Basic principle of Spectre attacks**
   We show the basic principle behind the different Spectre attacks. We discuss scenarios and assess how realistic they are.
3. **Remote Attacks**
   We show how microarchitectural timing differences can be measured over the network. This allows mounting timing attacks over the network.
4. **NetSpectre**
   We show how Spectre attacks can be combined with remote attacks, resulting in NetSpectre, a remote Spectre attack. We demonstrate that NetSpectre can be mounted in local networks, as well as across virtual machines in the cloud.
5. **Spectre Variants with different Microarchitectural Attacks**
   NetSpectre relies on Spectre attacks, a new class of attacks. We discuss different instances based on different microarchitectural attacks, besides the most trivial case of the Flush+Reload covert channel, showing that countermeasures targeting the cache are not complete.
6. **Countermeasures**
   We go into more detail on the countermeasures, some of which we already have seen ineffective.
7. **Challenges and Consequences**
   We discuss challenges in research and consequences on industry and academia which are caused by Spectre and therefore also NetSpectre.
8. **Closing remarks**

Takeaways

1. Speculative execution leaks internal secrets from software without bugs.
2. NetSpectre shows that Spectre attacks are not limited to local attackers but can be mounted via the network.
3. NetSpectre and Spectre attacks have a larger impact than assumed until this talk and they are difficult to mitigate and won't be fully mitigated soon.

Why Blackhat

Spectre attacks won't be easily fixed by operating system or hardware patches. Black Hat attendees need to learn what to consider about their own systems to stay safe.

List of Figures

List of Tables

Bibliography

[1] Al Framingdan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 526–540.

[2] Al-Haiqi, A., Ismail, M., and Nordin, R. A new sensors-based covert channel on android.

[3] Alagappan, M., Rajendran, J., Doroslovački, M., and Venkataramani, G. Dfs covert channels on multi-core platforms. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (Oct 2017), pp. 1–6.

[4] ARM. Arm neon instuctions. `https://developer.arm.com/technologies/neon`, 2018.

[5] ARM Limited. ARM Architecture Reference Manual. ARMv7-A and ARMv7-Redition., 2012. `https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf`.

[6] Bays, L. R., Oliveira, R. R., Barcellos, M. P., Gaspary, L. P., and Mauro Madeira, E. R. Virtual network security: threats, countermeasures, and challenges. *Journal of Internet Services and Applications 6*, 1 (Jan 2015), 1.

[7] Bosman, E., and Bos, H. Framing signals - a return to portable shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 243–258.

[8] Canella, C., Bulck, J. V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. *CoRR abs/1811.05441* (2018).

[9] Cateee. Disable branch prediction. `https://cateee.net/lkddb/web-lkddb/CPU_BPREDICT_DISABLE.html`, 2018. Accessed on 07.02.2019.

[10] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR abs/1802.09085* (2018).

[11] Cover, T., and Thomas, J. In *Elements of Information Theory* (2006).

[12] Dewan, M. C. Study of speculative execution and branch prediction. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.2934&rep=rep1&type=pdf`, 2006.

[13] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., and Halderman, J. A. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488.

[14] Engine, C. Capstone engine. `https://www.capstone-engine.org/`, 2018. Accessed on 01.08.2018.

[15] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Covert channels through branch predictors: A feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2015), HASP '15, ACM, pp. 5:1–5:8.

[16] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Piscataway, NJ, USA, 2016), MICRO-49, IEEE Press, pp. 40:1–40:13.

[17] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 693–707.

[18] Fog., A. The microarchitecture of intel, amd and via cpus. `https://www.agner.org/optimize/microarchitecture.pdf`, 2015.

[19] Fog., A. Test results for broadwell and skylake. `https://www.agner.org/optimize/blog/read.php?i=415#415`, 2018.

[20] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Cryptology ePrint Archive, Report 2016/613, 2016. `https://eprint.iacr.org/2016/613`.

[21] Genkin, D., Papadopoulos, D., and Papamanthou, C. Privacy in decentralized cryptocurrencies. *Commun. ACM 61*, 6 (2018), 78–88.

[22] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 217–233.

[23] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2017), SEC'17, USENIX Association, pp. 217–233.

[24] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: Long live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings* (2017), pp. 161–176.

[25] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 368–379.

[26] Gruss, D., Maurice, C., and Wagner, K. Flush+flush: A stealthier last-level cache attack. *CoRR abs/1511.04594* (2015).

[27] Gruss, D., Maurice, C., and Wagner, K. Flush+flush: A stealthier last-level cache attack. *CoRR abs/1511.04594* (2015).

[28] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 897–912.

[29] Hazen, J. Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer. `https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/`. Accessed on 06.01.2019.

[30] Hill, M. D., and Smith, A. J. Evaluating associativity in cpu caches. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1612–1630.

[31] Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.

[32] inside blog, L. Covert channels in internet protocols: A survey. `https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-3.html`, 2018.

[33] Intel. Intel advanced encryption standard(aes) new instructions set. `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`, 2010.

[34] Intel. Intel advanced vector instructions. `https://software.intel.com/sites/default/files/0d/2d/36884`, 2016.

[35] Intel. Intel® 64 and ia-32 architectures software developers manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[36] Intel. Speculative execution side channel mitigations. https://software.intel.com/sites/default/files/managed/5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[37] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 591–604.

[38] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses* (Cham, 2014), A. Stavrou, H. Bos, and G. Portokalidis, Eds., Springer International Publishing, pp. 299–319.

[39] Jang, Y., Lee, S., and Kim, T. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 380–392.

[40] Jayasinghe, D., Fernando, J., Herath, R., and Ragel, R. Remote cache timing attack on advanced encryption standard and countermeasures. In *2010 Fifth International Conference on Information and Automation for Sustainability* (Dec 2010), pp. 177–182.

[41] John L. Hennessy, D. A. P. *Computer Architecture: A Quantitative Approach, 4th Edition*, 4 ed. Morgan Kaufmann, 2006.

[42] Kay, J., and Pasquale, J. A performance analysis of tcp/ip and udp/ip. Tech. rep., Berkeley, CA, USA, 1992.

[43] Ketan Kulkarni, V. R. M. A review of branch prediction schemes and a study of branch predictors in modern microprocessors. https://www.researchgate.net/publication/266891966_A_Review_of_Branch_Prediction_Schemes_and_a_Study_of_Branch_Predictors_in_Modern_Microprocessors.

[44] Khan, A. K., and Mahanta, H. J. Side channel attacks and their mitigation techniques. In *2014 First International Conference on Automation, Control, Energy and Systems (ACES)* (Feb 2014), pp. 1–4.

[45] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. B. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *CoRR abs/1806.05179* (2018).

[46] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Rowhammer: Reliability analysis and security implications. *CoRR abs/1603.00747* (2016).

[47] Kiriansky, V., and Waldspurger, C. Speculative buffer overflows: Attacks and defenses. *CoRR abs/1807.03757* (2018).

[48] Kocher, P. Spectre mitigations in microsoft's c/c++ compiler. `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`, 2018.

[49] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).

[50] Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology — CRYPTO '96* (Berlin, Heidelberg, 1996), N. Koblitz, Ed., Springer Berlin Heidelberg, pp. 104–113.

[51] Kocher, P. C., Jaffe, J., and Jun, B. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (Berlin, Heidelberg, 1999), CRYPTO '99, Springer-Verlag, pp. 388–397.

[52] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (Baltimore, MD, 2018), USENIX Association.

[53] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium* (2016), T. Holz and S. Savage, Eds., USENIX Association, pp. 549–564.

[54] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[55] Llamas, D., and et al. Covert channels in internet protocols: A survey. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.5859`, 2005.

[56] LLVM. Introduce the "retpoline" x86 mitigation technique for variant 2. `https://reviews.llvm.org/D41723`. Accessed on 07.02.2018.

[57] LWN. Gcc 7.3 released. `https://lwn.net/Articles/745385/`. Accessed on 07.02.2018.

[58] Maisuradze, G., and Rossow, C. Speculose: Analyzing the security implications of speculative execution in cpus. *CoRR abs/1801.04084* (2018).

[59] MANTEL, H., AND SUDBROCK, H. Comparing countermeasures against interrupt-related covert channels in an information-theoretic framework. In *20th IEEE Computer Security Foundations Symposium (CSF'07)* (July 2007), pp. 326–340.

[60] MARCO GISBERT, H., AND RIPOLI, I. On the effectiveness of full-aslr on 64-bit linux. In-depth Security Conference 2014 (DeepSec) ; Conference date: 18-11-2014 Through 21-11-2014.

[61] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., BOANO, C. A., MANGARD, S., AND RÖMER, K. Hello from the other side: SSH over robust cache covert channels in the cloud. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017* (2017).

[62] O. ACIICMEZ, C. KOC, K. C. K., AND SEIFERT, J. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'07, Springer-Verlag, pp. 225–242.

[63] O. ACIICMEZ, S. G., AND SEIFERT, J. New branch prediction vulnerabilities in openssl and necessary software countermeasures. Cryptology ePrint Archive, Report 2007/039, 2007. `https://eprint.iacr.org/2007/039`.

[64] OPENGROUPS. Memset function. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/memset.html`, 1997. Accessed on 01.08.2018.

[65] OPENGROUPS. System socket. `http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.html`, 1997. Accessed on 01.08.2018.

[66] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology – CT-RSA 2006* (Berlin, Heidelberg, 2006), D. Pointcheval, Ed., Springer Berlin Heidelberg, pp. 1–20.

[67] PERCIVAL, C. Cache missing for fun and profit. In *Proc. of BSDCan 2005* (2005).

[68] PERLEBERG, C. H., AND SMITH, A. J. Branch target buffer design and optimization. *IEEE Trans. Comput. 42*, 4 (Apr. 1993), 396–412.

[69] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 565–581.

[70] RAICU, I., AND ZEADALLY, S. Evaluating ipv4 to ipv6 transition mechanisms. pp. 1091 – 1098 vol.2.

[71] REDHAT. Spectre gadget scanner. `https://people.redhat.com/~nickc/Spectre_Scanner/scanner.tar.xz`, 2018.

[72] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security 15*, 1 (Mar. 2012).

[73] Saha, S., Lawall, J., and Muller, G. Finding resource-release omission faults in linux. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2011), PLOS '11, ACM, pp. 1:1–1:5.

[74] Saltaformaggio, B., Xu, D., and Zhang, X. Busmonitor : A hypervisor-based solution for memory bus covert channels.

[75] Schwartz, J. The law of large numbers. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.555.5977&rep=rep1&type=pdf`.

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

[77] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. Netspectre: Read arbitrary memory over network. *CoRR abs/1807.10535* (2018).

[78] Security, P. Address-space-layout-randomization. `https://pax.grsecurity.net/docs/aslr.txt`, 2003.

[79] Shah, J. Understanding and study of intrusion detection systems for various networks and domains. In *2017 International Conference on Computer Communication and Informatics (ICCCI)* (Jan 2017), pp. 1–6.

[80] Smith, J. E. A study of branch prediction strategies. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (New York, NY, USA, 1998), ISCA '98, ACM, pp. 202–215.

[81] Song, D. X., Wagner, D., and Tian, X. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (Berkeley, CA, USA, 2001), SSYM'01, USENIX Association.

[82] Tankard, C. Advanced persistent threats and how to monitor and deter them. *Network Security 2011* (2011), 16–19.

[83] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium* (August 2018), USENIX Association. See also technical report Foreshadow-NG [88].

[84] van der Veen, V., Lindorfer, M., Fratantonio, Y., Pillai, H. P., Vigna, G., Kruegel, C., Bos, H., and Razavi, K. GuardION: Practical mitigation

of dma-based rowhammer attacks on arm. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2018)* (June 2018).

[85] van Eck, W. Electromagnetic radiation from video display units: An eavesdropping risk? In *Computers and Security* (1985), pp. 269–286.

[86] Wang, G., Chattopadhyay, S., Gotovchits, I., Mitra, T., and Roychoudhury, A. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR abs/1807.05843* (2018).

[87] Wang, Z., and Lee, R. B. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (Dec 2006), pp. 473–482.

[88] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report* (2018). See also USENIX Security paper Foreshadow [83].

[89] Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 159–173.

[90] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Oct 2019), vol. 00, pp. 428–441.

[91] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 719–732.

[92] Yeh, T.-Y., and Patt, Y. N. Alternative implementations of two-level adaptive branch prediction. *SIGARCH Comput. Archit. News 20*, 2 (Apr. 1992), 124–134.

[93] Zapechnikov, S., Miloslavskaya, N., and Tolstoy, A. Modeling of next-generation firewalls as queueing services. In *Proceedings of the 8th International Conference on Security of Information and Networks* (New York, NY, USA, 2015), SIN '15, ACM, pp. 250–257.

[94] Zero, G. P. Retpoline: A branch target injection mitigation - intel® software. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org`, 2018.

[95] Zheng, Y., Davis, B. T., and Jordan, M. Performance evaluation of exclusive cache hierarchies. *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004* (2004), 89–96.