Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning

Jonas Depoix
RheinMain University of Applied Sciences
Wiesbaden, Germany
jonas.depoix@student.hs-rm.de

Philipp Altmeyer
RheinMain University of Applied Sciences
Wiesbaden, Germany
philipp.b.altmeyer@student.hs-rm.de

ABSTRACT

The recently discovered Spectre vulnerabilities exploit design flaws in the architecture of modern CPUs and pose a threat to computer systems safety. In order to fix these vulnerabilities, changes to the architecture of current processors are necessary. Previous software mitigations are difficult to deploy and introduce considerable performance hits.

In this paper we present a real-time detection system, which identifies Spectre attacks by detecting cache side-channel attacks. Building upon previous research in the field of cache side-channel detection, we monitor Hardware Performance Counters to observe the CPUs cache activity and use a neural network to analyze the collected data. Since cache side-channels usually cause a very distinct cache usage pattern, our neural network is able to successfully identify a Spectre attack with an accuracy of over 99%, in our test environment.

CCS CONCEPTS

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

KEYWORDS

Cache Side-Channel Attacks, Spectre, Hardware Performance Counters, Machine Learning, Neural Networks, Real-time Detection

1 INTRODUCTION

The first practical implementation of a cache-based side-channel attack was presented in 2003 [52] and have evolved over the last couple of years in attacks such as EVICT+PRIME and PRIME+PROBE by Osvik et al. [43], or the more recent FLUSH+RELOAD attack by Yarom et al. [57]. Although these attacks have posed considerable threats in the past, cache-based side-channel attacks have just recently become even more relevant. This is due to the important role they play in making the Spectre and Meltdown exploits possible, which are currently having a disruptive impact on the way future CPU generations will be designed [31, 38].

While Spectre and Meltdown can only really be fixed by updating the CPUs hardware [31, 38], software solutions have been found, which are able to mitigate those attacks for the price of performance. In the case of Meltdown, KAISER was introduced by Maurice et al. [42] and implemented in Linux under the name of kernel-page table isolation (KPTI) [11]. Similar solutions have been implemented in Windows and Mac OS [27, 36]. While Spectre has proven to be a lot harder to mitigate, different solutions have been proposed for the individual spectre variants. Some solutions require editing the code of vulnerable software, which is a very costly, tedious and error-prone task [12]. Other solutions have been integrated into compilers like GCC and MSVC [40, 45, 53]. Therefore a recompilation is needed, to mitigate a software's vulnerabilities.

So in order for a user to be safe, he is required to update his operating system (Meltdown) as well as all of his software (Spectre), while being dependent on the publisher of these operating systems and software to actually provide such updates. Also the effectiveness of mitigations software publishers deploy is not always communicated transparently to the customers. Under these conditions it is likely that users are unable to update their software, simply forget to do so or are uncertain whether they are still vulnerable.

But Cloud providers and their customers are exposed to an even greater risk. Jann Horn has proven that Spectre variant 2 can be used to read memory of a guest VM running on the same KVM hypervisor as the attackers VM [25]. This means that a customer's VM is potentially vulnerable, even if its operating system and software is kept up to date, if the hypervisor or another VM running on the same last level cache (LLC) is outdated and therefore vulnerable. Since keeping the hypervisor and guest VMs up to date, is out of the control of a cloud providers customer, he has no way of being certain that his data is safe or making sure it is.

The same applies to the cloud providers. Although they should find themselves responsible for making sure their hypervisors are not vulnerable, they have no way of making sure that VMs are kept up to date. Therefore they can't prevent unpatched VMs from being a threat to other VMs.

These circumstances would make a potential real-time detection system of such attacks a valuable tool. Such a real-time detection system could identify an attacking process and terminate it immediately. Also a cloud provider could move a VM to an isolated machine, if it is suspected to have malicious intents. This way they can keep all VMs on a hypervisor save without shutting down a customer's machine. Thereby the consequences of a falsely detected attack are greatly reduced.

We believe that real-time detection of Spectre and Meltdown attacks will play a big role in keeping users safe, until these attack vectors can be shut down by proper hardware solutions. Therefore we developed a real-time detection system for Spectre using hardware performance counters and machine learning, which will be presented in this paper. Similar approaches have been used for

WAMOS2018, August 2018, Wiesbaden

real-time detection of cache-based side-channel attacks, such as FLUSH+RELOAD [2, 6, 8, 10, 14, 26, 41, 46, 51, 58, 59]. We build upon this research and apply the proposed ideas to detect Spectre attacks. We use Hardware Performance Counters to monitor the caching behaviour of all running processes and then use a neural network to identify malicious cache activity in the collected data. In previous research neural networks have proven to introduce a lot less false positives than heuristic approaches [10]. Since falsely identifying a process as malicious could result in this process being killed, keeping the amount of false positives as low as possible is essential.

This paper is organized as follows: in Section 2 we will cover some of the necessary background information to give a better understanding of the subject. We will explain the general idea of cache-based side-channel-attacks, specifically FLUSH+RELOAD (Section 2.1) and how Meltdown and Spectre work (Section 2.4). After explaining Hardware Performance Counters in (Section 2.5), we will briefly cover the basics of neural networks (Section 2.6) and look at which findings from previous research we can apply to our work (Section 2.7). In Section 3 we will explain our approach in greater detail, by illustrating the data set we used (Section 3.1) and how we implemented our real-time detection system (Section 3.2). The results we achieved using our approach will be covered in Section 4, followed by Section 5 and Section 6 in which we will discuss about our results and the potential they offer for future research.

2 BACKGROUND

2.1 Cache-based side-channel attacks

Side-channel attacks are attacks which do not directly exploit a weakness in the implementation of a computer system, but instead observe the side-effects which are generated by this implementation and use the observed data to conclude on the systems internal ongoings. This could be through various side effects, e.g. timing information, power consumption [32] or electromagnetic leaks [1].

Cache-based side-channel attacks, also known as cache-timing attacks, are types of side-channel attacks, which evolve around exploiting the fact that loading something from a CPUs cache is a lot faster than loading it from main memory. By timing how long it takes to access a specific memory address, an attacker can conclude whether the accessed data has already been in the cache or not. This side effect can be exploited in different ways and various attacks have made use of this.

The first practical implementation of a cache-based side-channel attack was presented by Tsunoo et al. in [52], where they successfully used cache timings to attack the Data Encryption Standard (DES). The EVICT+PRIME and PRIME+PROBE attacks have been introduced by Osvik et al. and were used to attack the Advanced Encryption Standard (AES) [43]. More recently the FLUSH+RELOAD attack by Yarom et al. [57] has seen a lot of use due to its simple implementation and efficient, fast and reliable results. It has seen applications in attacking various computations such as cryptographic algorithms [7, 28, 57], kernel addressing information [22], web server function calls [60] and user input [23, 37, 48].

As explained in Section 2.4, cache-based side-channel attacks also play an important role in making the Meltdown and Spectre attacks possible. Although it would be possible to use other type of cache-based side-channel attacks, the FLUSH+RELOAD attack is frequently chosen, as suggested by the original Meltdown and Spectre implementations [31, 38].

Due to the relevance of FLUSH+RELOAD we are going to mainly focus on this attack. In the following paragraphs we will explain the FLUSH+RELOAD attack in greater detail, to provide a better understanding of how this particular attack works, as well as cache-base side-channel attacks in general.

As per usual a FLUSH+RELOAD attack involves two parties. A victim and a spy process. The victim is performing an operation, while the spy tries to get information about what the victim is doing.

In order for the FLUSH+RELOAD attack to work in this case, three preconditions have to be met. First of all the spy has to be able to synchronize with the victim. Meaning that he has to start the attack as the victim starts the cryptographic operation. He also needs to have access to an instruction which allows him to evict a specific area from the CPUs cache. Usually this only is the case, if the instruction can be called with user-level privileges. But most importantly the CPU must have a mechanism like Kernel Same-page Merging (KSM) [4] or Transparent Page Sharing (TPS) [54] enabled [10].

KSM allows processes to share pages by merging different virtual addresses into the same page, if they reference the same physical address. It thereby increases the memory density, allowing for a more efficient memory usage. KSM was first implemented in Linux 2.6.32 and is enabled by default [33].

TPS is a proprietary technology of VMware and was developed with a similar purpose in mind. It also aims at making memory usage more efficient by sharing identical pages, while having the hypervisor managing the shared pages. But besides allowing processes inside of a VM to share pages, it also enables sharing pages between VMs. In this case cross-VM attacks using FLUSH+RELOAD become feasible.

This feature used to be enabled by default, but due to justified security concerns it is disabled as of VMware ESXi version 6 [55]. Also updates for all 5.x versions disable the feature, if it is enabled [55]. This however only disables sharing pages between different VMs, while pages are still shared within VMs [55]. So although cross-VM FLUSH+RELOAD attacks exploiting TPS are mitigated this way, attacks between processes running on the same VM still pose a considerable threat.

Consequently the spy and victim process could share memory pages under these circumstances. So if the victim accesses a memory address which is mapped by a shared page, it is saved to the cache. If the spy tries to access the same address afterwards, it is already in the cache, as it was just recently accessed by the victim. Since retrieving data from cache is significantly faster than fetching it from main memory, the spy can now tell whether the requested memory address was accessed by the victim, by measuring how long it took until it was retrieved.

While this allows the spy to find out if a memory address was accessed, he can't tell when it was accessed. This is where an instruction is needed, that allows to evict specific addresses from the cache. However most modern Intel processors offer the `CLFLUSH` assembly mnemonic [44]. It is available on their Core i3, i5, i7 and

Xeon models and can be executed from user-level, which allows it to be run by an unprivileged process. By calling the `CLFLUSH` mnemonic with a memory address, the entire cache line which includes the content referenced by the address is evicted from the cache. Intel CPUs use a inclusive hierarchy, meaning that the caches of a certain level contain the content of all caches with a lower level than theirs. This entails that flushing an address from the LLC, also flushes it from all other cache levels [10].

This mechanism can be exploited by the spying process, to make sure that the memory addresses it is spying on are not already in the cache. This could be implemented by using the algorithm shown in Algorithm 1.

ALGORITHM 1: FLUSH+RELOAD [10]

Data: 0xABC is a physical address in a page shared by the spy and the victim. FLUSH_FREQUENCY is the frequency in which the spy is checking if the victim has accessed 0xABC. CACHE_ACCESS_THRESHOLD is the maximum amount of time it takes to get data from cache instead of main memory.

```
1  while spy is attacking do
2      clflush(0xABC);
3      sleep(FLUSH_FREQUENCY);
4      /* victim may or may not access 0xABC while the spy
          is sleeping                                      */
5      start_time = get_timestamp();
6      load(0xABC);
7      end_time = get_timestamp();
8      if end_time - start_time < CACHE_ACCESS_THRESHOLD then
9          /* victim has accessed 0xABC since last clflush */
10     else
11         /* victim most likely has not accessed 0xABC since
              last clflush                                 */
12     end
13 end
```

By regularly evicting the relevant memory addresses from the cache and accessing them after waiting for a given time interval (lines 2-6), the attacker can then tell if these addresses were accessed by the victim in the meantime (lines 7-12).

2.2 Out-of-Order Execution

Modern processors use out-of-order execution to maximize the utilization of all execution units of a CPU core. Rather than processing the instructions in the sequential program order, the CPU can execute subsequent instructions in parallel or even before preceding instructions. While one execution unit is busy or waiting for required resources, other execution units can run different instructions. When an instruction has been completed, it is queued in a reorder buffer. Once all preceding instructions have been executed, the instructions are committed and cleared from the reorder buffer. Eventually the instructions are retired in the specified program execution order [31, 38].

2.3 Speculative Execution

Speculative execution is widely used among several CPU microarchitectures to increase performance. When the control flow of the application depends on the result of a preceding instruction, the processor can predict the most likely path of the program and speculatively execute the next instructions. Depending on the size of the reorder buffer, speculative execution can run several hundred instructions ahead [31].

When using out-of-order and speculative execution, the processor cannot immediately determine the next instruction to execute. This can for example occur when the control flow depends on an uncached value in the physical memory, in case of a conditional or unconditional branch. Because this memory is much slower than the internal CPU registers, it can take several hundred clock cycles before the value is fetched. Instead of waiting for the value to arrive, the processor guesses the future path that the program will follow and speculatively executes instructions along the predicted path. This optimization method is called branch prediction. When the value requested from the external memory arrives, the CPU compares it with its guess. If the predicted path was wrong, the CPU discards the incorrectly executed instructions. This results in a performance equal to idling. But if the predicted path was right, the speculatively executed instructions lead to a significant performance gain [31].

A second example for speculative execution is the delay that occurs by translating the virtual memory addresses of a process to physical memory addresses. In addition to translating the memory addresses, the CPU also checks if the process has the permission to access to the requested virtual addresses. While the processor is waiting for the result of the permission check, it can speculatively execute the read and the following instructions. If the process has insufficient permissions, the CPU raises an exception and the results of the speculatively executed instructions are reverted. But similar to the aforementioned branch prediction, if the process has access to the read memory address, the speculatively executed instructions add to an increased performance [38].

Speculative execution can lead to execution of a program in incorrect ways, but the CPU is designed to revert the results of incorrect speculative executions. Therefore these errors were assumed to be safe prior to the Meltdown and Spectre attacks. But it turns out that not all side effects of speculative execution are reverted and some previously leaked information, e.g. cache contents, can survive the CPU state revision. The Spectre and Meltdown attacks exploit this flawed behaviour by recovering this leaked information from the cache [31].

2.4 Meltdown and Spectre

In [31] Kocher et al. presented two variants of the Spectre attack which exploit the prediction of conditional and unconditional branches. Meltdown [38] is a related attack which does not rely on branch prediction but exploits the out-of-order execution of instructions. When an instruction raises an exception, subsequent instructions are speculatively executed, before the exception is handled.

Meltdown relies on a vulnerability specific to Intel and ARM processors and can be mitigated by the implementation of KAISER

[42] in operating systems. On the contrary Spectre applies to vastly more CPU architectures and cannot be mitigated as effectively [31]. Because of these limitations we focus our work on the detection of Spectre attacks.

Spectre variant 1 exploits the prediction of conditional branches. The simplified example in Listing 1 shows a conditional branch that receives an unsigned integer `x` as an input. This code could be part of a function in a system call or a library where `x` is controlled by an untrusted source. In this example `array_size` is assumed to be the size of `array1` [31].

LISTING 1: Conditional Branch Example [31]

```
1 if (x < array_size)
2       y = array2[array1[x] * 4096]
```

To ensure that a malicious `x` does not access memory outside the range of `array1` the code does a bounds check. This check is crucial because an out of bounds access could trigger an exception or reveal sensitive data. In case of normal execution this program flow causes no security risks. However during speculative execution the read of `array1` could be performed before the result of the bounds check on `x` is known. As previously explained, this could happen when the value of `array_size` is not in present in the CPU cache and has to be fetched from external memory. Because the effects of the speculative read on the cache state are not reverted, an attacker could use a side channel to recover the content of the accessed memory location [31].

To perform the attack, an adversary has to run the example code in such a way that the value of a malicious `x` is selected, so that `array[x]` resolves to a secret byte `k` somewhere in the victim processes memory. Further `array_size` and `array2` have to be evicted from cache, but `k` is cached. To mistrain the CPU branch prediction, the adversary runs the code beforehand multiple times with valid values for `x`, leading the branch predictor to expect the `if` condition to be true [31].

When `array_size` is evicted from cache, reading the value results in a cache miss and causes a considerable delay before the result arrives from external memory. While one execution unit is busy waiting for the outcome of the branch condition, the CPU speculatively executes the next instructions. Because the branch predictor has been mistrained earlier to assume the condition is likely to be true, the speculative execution logic adds `x` to the address of `array1` and requests the resulting address (the location of the secret byte `k`) from memory. Since `k` is assumed to be cached during the attack, the read quickly returns the value of `k`. Subsequently the speculative execution calculates the address of `array2[`$k * 4096$`]` and attempts to read this address from memory. In the meantime the result of the branch condition may be determined at last and the processor reverts the register state due to the incorrectly speculated branch. But the speculative read from `array2` leaves traces in the cache state, depending on the address of the secret byte `k` [31].

To restore the value of `k`, the adversary determines which location in `array2` was loaded into the cache. Because the speculative execution cached `array2[`$k * 4096$`]`, the value of the secret byte `k` can be exposed using a cache side channel like FLUSH+RELOAD or PRIME+PROBE [31].

In addition to this example, Spectre variant 1 can exploit many different instruction patterns. Alternatively to the bounds check, the conditional branch could be checking a more complex safety result or an object type. Likewise the speculatively executed code could be implemented with a larger amount of instructions or could use a different method to leak the secret byte, e.g. writing a comparison result to a fixed memory location [31].

Instead of exploiting the speculative execution of conditional branches, Spectre variant 2 works by poisoning the prediction of indirect branches. When the address of an indirect branch cannot be resolved immediately, for example because of a cache miss that causes a delay, speculative execution will jump to a predicted address to continue execution. Much like the conditional branch prediction, the predicted address depends on locations taken by previous code executions [31].

So to perform an attack in Spectre variant 2, the adversary mistrains the branch predictor by jumping to malicious locations in the attacker process. Although the branch predictor is trained on the context `A` of the attacker process, the CPU makes its prediction in context `B` on the basis of training data from context `A`. Hence the adversary can misdirect speculative execution to jump to locations that would not be reached during normal program execution. This implies that arbitrary code mapped in the victims address space can be executed [31].

Since the speculative execution has side effects, e.g. the traces in the cache state exploited by Spectre variant 1, it is possible to read the memory of the victim process. In order to leak the information via a side channel, the attacker needs to locate a so-called Spectre gadget. This Spectre gadget is a code fragment, that transfers the victim's information through the side channel. This gadget could be found in a shared library that is mapped into the victim's process, without having to search in the victim's own code [31].

Depending on what state is known and can be controlled by the attacker, or where the secret information is located, plenty of other attacks are also feasible. Also for specific gadgets control over a single register, value on the stack, or memory value is sufficient for an attack [31].

2.5 Hardware Performance Counters

Most modern microprocessors are equipped with special purpose registers called Hardware Performance Counters (HPCs). These are used to count the occurrences of different kind of CPU events, e.g. clock cycles, cache hits and cache misses for each cache level or branch misses. Applications can attach to these counters of a specified event type and read the counters of a given process, thread or the entire CPU. A HPC is increased each time an event of the relevant type occurs and usually is reset to zero after its value has been read.

A common use-case for HPCs is performance profiling, where detailed information such as caching behaviour can be very valuable [3]. But as previous work has shown (see Section 2.7), they can also provide a useful metric for detecting side-channel attacks by identifying malicious cache activity. Since HPCs are implemented into the processors architecture, they can be used with an insignificant performance overhead, which makes them a good fit for real time detection. Also they can be accessed from user-level with no

privileges needed, unless the process or thread which is monitored runs with higher privileges. Therefore a detection system using HPCs wouldn't need to be run by a privileged user, which usually is favorable.

A widely used interface to Hardware Performance Counters is the Linux command-line tool `perf` [13]. This tool allows to collect, visualize, filter and aggregate data gathered through the HPCs [10]. It contains the sub-command `perf-stat` which can be used to monitor CPU events of a specified type selectively system-wide, for a target process or a target thread.

`PAPI` (Performance Application Programming Interface) is a library which provides a unified interface to Hardware Performance Counters for all CPU models (which support HPCs). While not all CPUs support the same HPCs, those collecting the same information at can be addressed using the same name.

An advantage `PAPI` has over `perf-stat` is that it has a more finely grained resolution. While `perf-stat` allows for taking multiple samples a second, the smallest interval between two consecutive samples is 100 ms [10]. On the other hand applications using `PAPI` have been used up to a maximum resolution of 3 μs, making it more than 30000 times faster [9]. Although this difference might not be as crucial for performance profiling, it can be for detecting side-channel attacks.

2.6 Neural Networks

Artificial neural networks are machine learning models inspired by the structure of the human brain. Because they are particularly good at recognizing patterns in high-dimensional data, neural networks have proven to be very effective at solving classification tasks [35].

The goal of classification is to specify which category a given input belongs to. In the case of detecting Spectre attacks the input is the collected HPC data of different processes and the two output categories are *benign* and *malicious*. Thanks to the neural networks ability to learn nonlinear relations of the input set, it is possible to reliably classify complex data without the need for manually crafted features [21].

Neural networks consist of numerous artificial neurons arranged in multiple layers. Each neuron has a value and weighted connections to neurons in the subsequent layer. The simplest network architecture is a feedforward network. Figure 1 shows a feedforward network with an input and output layer and one hidden layer. Each node in the output layer corresponds to a category. To predict a value, the network takes the inputs from the input layer, feeds it to the hidden layer and outputs the prediction at the output layer. The output node with the highest value is the predicted category. In this example the layers are *fully connected*, because each neuron of one layer is connected to every neuron of the previous layer.

To predict a result based on a given input, each neuron sums the weighted values received from all connected neurons of the previous layer and passes the result through an activation function. This activation function squashes the resulting sum to a defined range, usually between 0 and 1. The sigmoid [24] function is often used as the activation function in neural networks. In order to learn the correct mapping between input and output values, the weight of each neural connection has to be adapted.

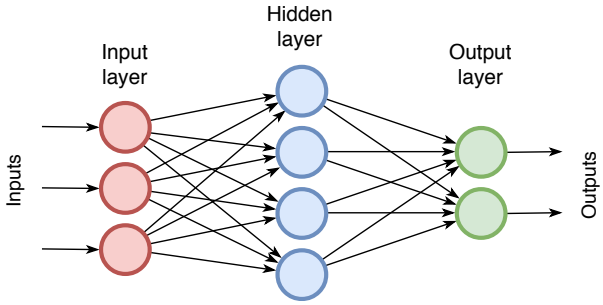

Figure 1: Simple feedforward neural network

The learning algorithm used for fitting the weights of the neural network is called backpropagation. During the training phase, the network receives an input and predicts an output value. Then the deviation between the output and the expected value is computed. This error value is propagated backwards from the output layer to the input layer and the weights of each node are updated accordingly [21].

Thus neural networks rely on labeled training data to calculate the error of the prediction, unlike unsupervised machine learning methods. To achieve good results, a large training set is necessary [35].

2.7 Related Work

Previous research has already shown the potential HPCs have for detecting side-channel attacks.

In [10] Chiappetta et al. used HPCs to detect FLUSH+RELOAD attacks on RSA, AES and ECDSA. They implemented a daemon constantly monitoring HPCs for the number of total instructions, total CPU cycles, L2 cache hits, L3 cache misses and L3 cache total accesses [9]. Using this data they presented and compared three different methods for detecting ongoing FLUSH+RELOAD attacks. One approach used was correlation-based, while the other two were different machine learning techniques, with one of them being unsupervised and the other being supervised, using a neural network.

To train the neural network they collected data of the relevant HPCs for different kind of processes. Besides collecting data of processes running FLUSH+RELOAD attacks, they also collected data of processes running common applications like an Apache web server.

While all techniques were able to detect an attack in most cases, the machine learning approaches did cause a lot less false positives. Also the neural network approach did prove to be the most resilient when the data was noisy. This way it could more accurately detect attackers, even if the attacker tried to additionally perform unsuspicious operations to obfuscate his intentions.

Bazm et al. built upon the research of Chiappetta et al. [10] and proposed a similar solution in [6], which specifically tries to detect cross-VM side-channel attacks in an IaaS environment. They used the Gaussian anomaly detection method to analyze the data collected by the HPCs, achieving promising results. Related research focusing on detecting side-channel attacks in cloud environments has been done by Zhang et al. in [58] and Inci et al. in [26].

Another interesting application of HPCs for malware detection was proposed by Alam et al. in [2]. They also used machine learning techniques to detect the WannaCry ransomware [16] by analyzing HPC data. They were able to decrease the amount of false positives by using recurrent neural networks (RNN) with long short-term memory (LSTM) cells. These kind of networks especially excel at processing sequential data [20].

3 APPROACH

As explained in Section 2.4, Spectre is only possible through the combination of two requirements. Firstly the attacker needs to be able to access data in speculatively executed instructions, which would not be accessible during correct program execution. Secondly he needs to be able to leak the accessed data through a side-channel. Most of the mitigations which have been introduced so far mainly focus on mitigating Spectre by trying to shut down the first requirement. But this has proven to be very hard to do and impossible without considerable performance hits [49]. Therefore we introduce a solution which prevents Spectre attacks of the variants 1 and 2 by stopping the attacker from leaking the accessed data through a side-channel. If the attacker is not able to leak the accessed data, the fact that this data can be accessed during speculative execution effectively no longer poses a threat.

To do this we built upon the work of Chiappetta et al. in [10], which was covered in Section 2.7. We utilize the fact that every cache side-channel attack has observable side effects. To execute a FLUSH+RELOAD attack for example, the attacker needs to constantly flush cache lines and check if the memory has been accessed since the last flush. This means that the attacker will have to do a lot of cache accesses, of which a lot will be cache misses, in a repetitive pattern. By constantly monitoring the HPCs (Section 2.5) of a process, we therefore can reliably predict if the attacker is accessing the cache in a malicious way.

As suggested by Chiappetta et al., we use a neural network trained to find malicious activities in the collected HPC data [10]. The data set we used to train this neural network is explained in greater detail in the following section.

3.1 Data set

In order to train the supervised learning model, we created a data set consisting of HPC data collected from various benign processes and malicious Spectre implementations. These data points are respectively labeled as benign or malicious. Our approach uses performance counters attached to each process instead of accumulated readings of the entire CPU. This separation allows the model to classify each process as benign or malicious. A detection system can then take actions per process based on the predictions of the neural network. For instance the system can notify the user when an application is behaving suspiciously or kill a malicious process.

Since only a small number of performance counters can be monitored simultaneously, we selected three processor events based on the run time characteristics of the Spectre implementations. These three events are the L3 cache misses (`L3_TCM`), L3 cache accesses (`L3_TCA`) and total number of instructions (`TOT_INS`).

The L3 cache misses event (`L3_TCM`) appears to be a good indicator for detecting cache side-channels and hence Spectre attacks. As explained in Section 2.1, cache side-channels like `FLUSH+RELOAD` operate by frequently flushing a specific chunk of memory from the cache and measuring the access times of a memory read operation. Therefore the adversary process shows significantly higher cache miss rates. Flushing the cache with the `CLFLUSH` instruction propagates to all cache levels. Thus inspecting the last level cache (i.e. L3 cache) can identify intentional cache evictions.

In addition to the L3 cache misses, we chose the L3 cache accesses (`L3_TCA`) as a reference point for total cache activity. A process with a higher number of cache accesses presumably has a higher rate of cache misses. So to prevent a benign process with a large number of cache misses to be detected as a false positive the neural network learns a relation between cache misses and total cache accesses.

The total number of instructions (`TOT_INS`) was selected to account for the workload the monitored process puts on the CPU in relation to the number of cache misses. Because a malicious process typically has a short loop that repeatedly attacks a victim process, the percentage of cache misses in relation to the total number of executed instructions is likely to be higher than the rate of a benign application.

To generate the data set, we considered the following eleven scenarios:

(1) *Wordpress*: PHP based CMS with nginx as web server and MariaDB as database server [18, 19, 50]
(2) *Ghost*: Node.js based CMS with nginx as web server and MariaDB as database server [17, 29]
(3) *stress -c*: one worker process spinning on `sqrt()` [56]
(4) *stress -m*: one worker process spinning on `malloc()`/`free()` [56]
(5) *stress -i*: one worker process spinning on `sync()` [56]
(6) *Chrome*: user doing light web browsing [39]
(7) *SpectrePoc*: implementation of the Spectre variant 1 code presented in [31]
(8) SpectrePoc no `CLFLUSH`: SpectrePoc without the usage of `CLFLUSH`
(9) spectre-chrome: Spectre implementation in JavaScript [5]
(10) Spectre Check: Spectre vulnerability check for web browsers, implemented in JavaScript [34]
(11) Spectre Cross-Process: Spectre variant 2 cross-process read demo [15]

The first two scenarios are examples of server workloads. To get HPC data of a process under load, the homepages of both content management systems were repeatedly queried with 50 requests per second. The next three scenarios cause a high load on the system, but are classified as benign. The *Chrome* scenario is a representation of a casual desktop workload. The remaining five scenarios are sample implementations of Spectre variant 1 and 2. Because all common browsers have deployed updated versions with Spectre and Meltdown mitigations, the support for `SharedArrayBuffer` had to be re-enabled in Chrome to be able to execute the JavaScript based attacks. For each scenario the three aforementioned processor events were recorded separately for all corresponding processes for sixty seconds, with a precision of 100 milliseconds. Overall we collected a total of 15635 data points.

Figure 2 depicts the total L3 cache misses of the ten processes with the highest cache miss rate. The processes associated with

scenario 1 are *php-fpm7.1_2*, *php-fpm7.1_3* and *php-fpm7.1_4*. Respectively *node* corresponds to scenario 2, *stress_m* to scenario 4 and *chrome_browsing* to scenario 6. The plotted Spectre attacks *spectre*, *spectre_noflush*, *spectre_chrome* and *spectre_check* are the recorded processes for the scenarios 7 to 10. As expected, the plot indicates that the number of total cache misses is significantly higher for the Spectre processes. However the *node* process also shows a high rate of cache misses.

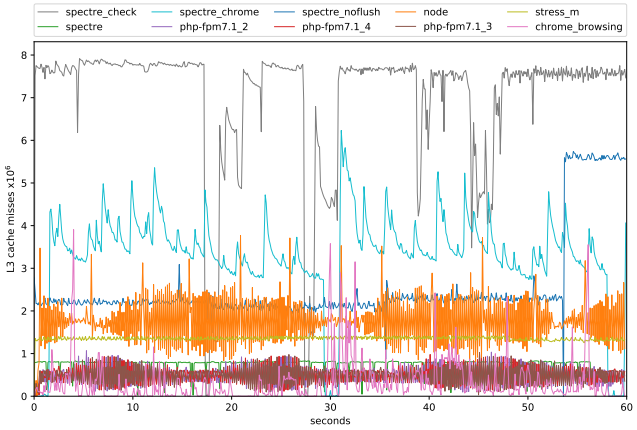

Figure 2: L3 Total Cache Misses

Figure 3 shows the total number of L3 cache accesses. The scenario associated with *spectre_attack* is scenario 11. Similar to Figure 2, the Spectre attacks have a high number of total cache accesses. Because of the usage of a different cache eviction method, *spectre_noflush* has an exceptionally high count. The second highest number has the *Node.js* process. This indicates that a high memory activity correlates with a large number of cache misses.

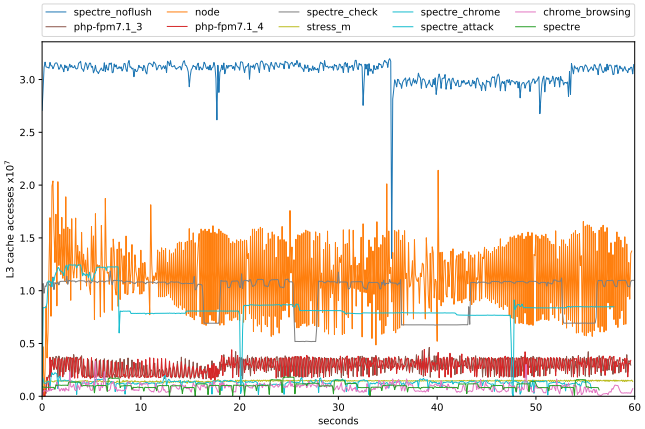

Figure 3: L3 Total Cache Accesses

Lastly the accumulated number of total instructions are illustrated in Figure 4. The *stress_c* process corresponds to scenario 3, which creates a high CPU load and causes large numbers of total instructions executed. But since the *stress_c* has minimal cache accesses, the neural network can learn to classify similar CPU intensive tasks as benign. As already depicted in Figure 2, both *spectre_check* and *spectre_chrome* have a high cache miss rate. Combined with the large number of total instructions, the relation between cache misses and executed instructions is significantly higher than benign processes.

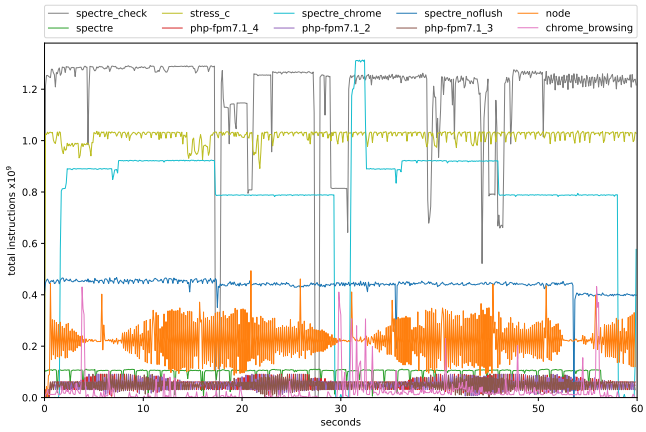

Figure 4: Total Instructions

These results show that the number of L3 total cache misses, L3 total cache accesses and total instructions are good indicators for identifying cache side-channel attacks.

3.2 Implementation

Our detection system consists of three different services, which each run in independent processes. An overview of the systems architecture is illustrated in Figure 5 and will be discussed in this section.

The role of the `ProcessLifecycleService` is to track which processes are started and stopped by the operating system, so that we can immediately start monitoring these processes for malicious behaviour. This is done using `netlink`. `netlink` is a socket-based Linux kernel interface used for communication between kernel and user-space processes [30]. The `ProcessLifecycleService` opens a socket to this interface, to be notified when a process is started or stopped. The PIDs of relevant processes, and information about whether the detection system should start or stop watching them, are then forwarded to the next service through a pipe.

The `HPCService` takes care of watching the HPCs of the processes, it receives from the `ProcessLifecycleService`. This is done using `PAPI`, as explained in Section 2.5. `PAPI` provides a specific data structure for this, which can be allocated by the `HPCService` and attached to the HPCs of choice. It then takes care of writing the current values of the attached HPCs into this data structure. Every 100 milliseconds the `HPCService` reads and resets all attached counters from this data structure. The PIDs of the watched processes with their corresponding HPC values are then piped to the next service, after each 100 millisecond interval.

The actual detection of potentially ongoing side-channel attacks is done by the `SCADetectionService`. It uses the HPC data it receives from the `HPCService`, to predict whether the corresponding

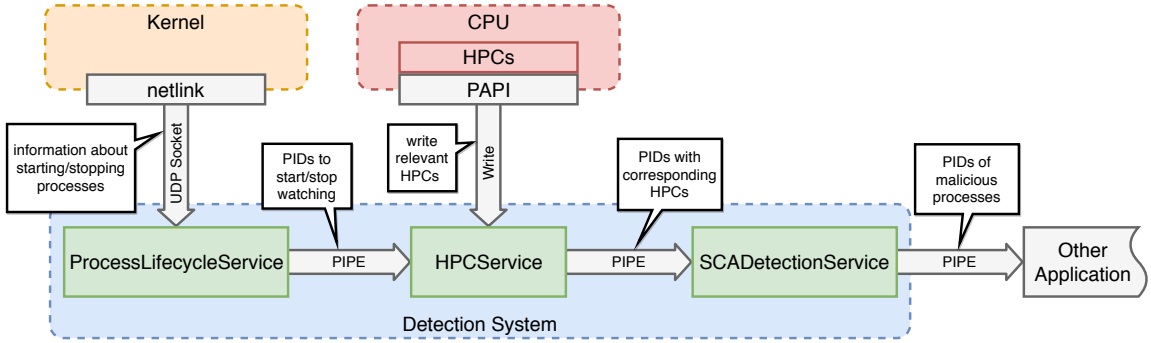

Figure 5: Application architecture

process is behaving maliciously. The prediction is done by a feed-forward neural network, which was previously trained on the data set explained in Section 3.1.

This neural network has three input neurons, which correspond to each of the three collected HPCs. It has one fully connected hidden layer with 32 neurons and one output neuron. The output neuron uses a sigmoid activation function, which maps the output to a value between 0 and 1 [24]. The examined process is considered malicious if this output is above 0.5 and unharmful otherwise. While we have tried different network architectures, this one has proven to give the best results, without inducing overfitting, using our data set.

The PIDs of malicious processes could then potentially be piped to another application, which decides what to do with those processes. As of now our application only focuses on identifying attacks and does not dictate how to deal with them, as discussed in Section 5.

4 EXPERIMENTS AND RESULTS

After training our detection systems neural network with the data set explained in Section 3.1, we collected data to measure its performance, which we will discuss in this section.

We split up 10% of our data set, which we did not use for training, to be able to validate our trained network with data it hasn't seen before. This makes a validation set with a total number of 1564 data points.

The two metrics we used as main indicators for the performance of our neural network, are the prediction accuracy and the F-score [47]. Also we took a close look at the amount of true positives, false positives, true negatives and false negatives, which are common metrics for binary classifiers.

Table 1 gives an overview of the exact metrics we collected during the validation of our neural network. Also Figure 6 illustrates the ground truth and the predictions of our neural network. It also shows the population of the classes true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

While still being very good, it is noticeable that the accuracy for detecting positives is not as good as for detecting negatives. Meaning that identification of benign processes is actually more accurate than detection of malicious processes. While this may seem like a rather undesirable result, looking at Figure 6 should make clear, that this is not as unfavorable as it might seem at first sight. Besides the accuracy for detecting positives still being above 97%, which can be considered fairly reliable, it should be kept in mind, that these metrics are calculated only looking at individual data points. But Figure 6 illustrates, that false negatives (as well as false positives) always are individual outliers and never happen

total number of datapoints	**1564**
number of positives	317
number of negatives	1247
accuracy (total)	**99.23%**
accuracy (positives)	97.16%
accuracy (negatives)	99.67%
F-score	**0.9716**
number of true positives (TP)	308
number of false positives (FP)	4
number of true negatives (TN)	1243
number of false negatives (FN)	9

Table 1: Validation results

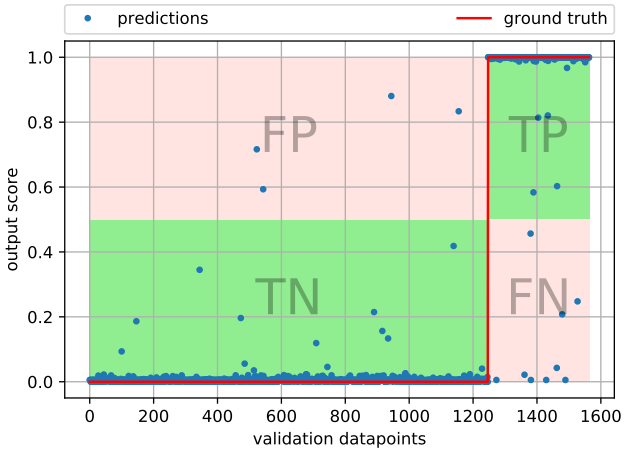

Figure 6: Validation results

consecutively. In practice this means that even if a malicious process can't be successfully detected with the first set of data we collect from its HPCs, it should be detected within the next cycle, after 100 milliseconds.

While arguably there still could be some damage done within these additional 100 milliseconds, the amount of false positives could prove to be more of a problem in practice. Even though it is lower than the amount of false negatives, the consequences of a falsely predicted positive could be more devastating, since it could lead to that process getting killed. Immediately killing a malicious process could only be considered as a valid countermeasure, if our system does not have any false positives at all. The chance of mistakenly killing any process at any time, would pose to much of a threat to any kind of system to make our detection system feasible.

To address these problems we discuss some ideas which could be implemented to decrease the rate of false negatives, as well as false positives, in Section 6.

In our experimental setup with an Intel Core i7-7820HQ processor and 32GB of ram, the detection system has a CPU usage of $4 - 5\%$, while using a polling rate of 100ms. Section 6 also covers actions which could be taken to improve our detection systems performance.

5 DISCUSSION

Looking at the results in Section 4, we believe that real-time detection of Spectre attacks is definitely a feasible protection method. Even more so with the implementation of the improvements suggested in Section 6. This poses the question whether preventing such attacks could be done more effectively using real-time detection tools, instead of struggling to implement mitigations which severely hurt performance, although it is known that the underlying problems only can really be solved by updating the hardware.

One thing which could be holding our detection system back from being a considerable alternative to the software mitigations, is that a malicious process has to do something malicious first, before it can be identified as such. Since we read out a processes HPCs every 100 milliseconds, it can take up to 100 milliseconds to identify it as malicious. According to [25] Spectre variant 1 can readout about 2000 bytes per second on a Intel Haswell Xeon CPU. So if the attacker would be identified after 100 milliseconds under these circumstances, he would have already read 200 bytes. Whether this is enough data for the attacker to do some damage, depends on the context of the attack. In Section 6 we suggest measures to make this less of a problem.

Ultimately it is up to the user to decide whether the protection provided by a detection system is enough, to make it a viable alternative for him. But even if it doesn't make for an alternative, it definitely makes for a valuable supplement. A detection system allows for keeping legacy software safe, even if the publisher no longer provides updates. Therefore a combination of a detection system and software mitigations, will be the safest option for the user.

As mentioned in Section 3.2, our detection system does not dictate how to handle malicious processes, after they have been identified. The event is piped to another application, which then takes care of this event. There are different options on how this hypothetical application could handle this event. The most obvious option is to just kill the process. But as mentioned before, this can lead to a process accidentally being killed, if the detection system falsely predicts it to be malicious. Therefore the safest option would be to halt the process and let the user decide, what to do with it. This would make sure that no processes are killed accidentally, but in practice this only is feasible on a desktop system. A good combination of those two options would be to halt the process first and continue it after a set amount of time, which is long enough to disrupt a potentially ongoing cache attack. If it is identified as malicious again, it could then be killed. Also it should be made sure that the executable which the process was running can't be started again, to prevent the attacker from eventually still being able to execute his attack in multiple 100 millisecond time windows.

6 CONCLUSION AND FUTURE WORK

In this paper we introduced a real-time detection system for Spectre attacks. It identifies malicious processes by monitoring their Hardware Performance Counters and analyzing this data using a neural network. With this technique we were able to achieve a detection accuracy of over 99%, which shows the potential that Hardware Performance Counters and Machine Learning offer for detecting side-channel and thereby Spectre attacks.

Although we were able to achieve good results, there is still room for improvements, which future work could build on.

First of all our detection system was implemented as a proof of concept and therefore isn't optimized for performance as much as it could be. As of now the 100 millisecond HPC polling rate was chosen, as it has proven to work well without hurting the performance of the machine it was running on. If the detection system itself would run more efficiently, a higher HPC polling rate would become feasible. This would also address a lot of the issues discussed in Section 5, as a higher polling rate would mean less time will pass until an attacker is identified.

Also our neural network should be trained on more diverse implementations of Spectre attacks, specifically ones using different kinds of side-channel attacks. Since different types of side-channel attacks have distinctive cache usage patterns, a Spectre attack using a side-channel which the neural networks has never seen before, could potential stay undetected. This would reduce the amount of false negatives and lead to more reliable predictions.

Also a broader data set of benign HPC data could be collected for training, which could decrease the amount of false positives. But even if the number of false positives in the validation set is 0, it is impossible to completely rule out that false positives will ever happen in practice. However only killing a process after it has proven to be malicious repeatedly, as suggested in Section 5, can effectively nullify the risk that false positives bring.

If our detection system is also applicable to Meltdown attacks, is a question which could be picked up by future research. But since Meltdown also uses cache side-channels to leak the maliciously collected data in the same way Spectre does, our detection system should also be able to detect Meltdown attacks. However we haven't done any experiments yet, to back this theory up.

As explained in Section 1 Spectre poses a significant threat in a cross-VM scenario. Therefore doing further research on how well

our detection performs running on a hypervisor, could prove it to be a great tool for cloud providers to keep their customers safe. In [6] and [58] Bazm et al. have successfully applied the ideas from [10] to such a cross-VM scenario. Since we also built our implementation based on concepts introduced in [10], we are confident that our system would also be feasible for detecting cross-VM attacks. This however would have to be confirmed by future research.

REFERENCES

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2002. The EM side-channel (s). In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 29–45.

[2] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. 2018. RAPPER: Ransomware Prevention via Performance Counters. *arXiv preprint arXiv:1802.03909* (2018).

[3] Glenn Ammons, Thomas Ball, and James R Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices* 32, 5 (1997), 85–96.

[4] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.

[5] ascendr. 2018. spectre-chrome. https://github.com/ascendr/spectre-chrome. (2018).

[6] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Sudholt, and Jean-Marc Menaud. 2018. Cache-Based Side-Channel Attacks Detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In *Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on*. IEEE, 7–12.

[7] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 75–92.

[8] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2017. CacheShield: Protecting Legacy Processes Against Cache Attacks. *arXiv preprint arXiv:1709.01795* (2017).

[9] Marco Chiappetta. 2015. quickhpc. https://github.com/chpmrc/quickhpc. (2015).

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* 49 (2016), 1162–1174.

[11] Jonathan Corbet. 2017. The current state of kernel page-table isolation. (2017). https://lwn.net/Articles/741878/

[12] Intel Corporation. 2018. Speculative execution side channel mitigations. (May 2018).

[13] Arnaldo Carvalho De Melo. 2010. The new linux 'perf' tools. In *Slides from Linux Kongress*, Vol. 18.

[14] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 559–570.

[15] Theodore Dubois. 2018. Spectre Cross-Process Read Demo. https://github.com/tbodt/spectre. (2018).

[16] Jesse M Ehrenfeld. 2017. Wannacry, cybersecurity and health information technology: A time to act. *Journal of medical systems* 41, 7 (2017), 104.

[17] Ghost Foundation. 2018. ghost. (2018). https://ghost.org

[18] MariaDB Foundation. 2018. MariaDB. (2018). https://mariadb.com

[19] WordPress Foundation. 2018. wordpress. (2018). https://wordpress.org

[20] Felix A Gers, Douglas Eck, and Jürgen Schmidhuber. 2002. Applying LSTM to time series predictable through time-window approaches. In *Neural Nets WIRN Vietri-01*. Springer, 193–200.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[22] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 368–379.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *USENIX Security Symposium*. 897–912.

[24] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. In *Neural networks for perception*. Elsevier, 65–93.

[25] Jann Horn. 2018. Reading privileged memory with a side-channel. (2018). https://support.google.com/faqs/answer/7625886

[26] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2016. Co-location detection on the cloud. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 19–34.

[27] Alex Ionescu. 2018. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2018). https://borncity.com/win/2018/01/03/design-flaw-in-intel-cpus-set-operating-systems-at-risk/

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

[29] Joyent. 2018. Node.js. (2018). https://nodejs.org

[30] Michael Kerrisk. 2018. netlink. (2018). http://man7.org/linux/man-pages/man7/netlink.7.html

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

[32] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual International Cryptology Conference*. Springer, 388–397.

[33] KVM. 2015. KSM — KVM,. (2015). https://www.linux-kvm.org/index.php?title=KSM&oldid=173356

[34] Tencent's Xuanwu Lab. 2018. Spectre Vulnerabilty Check. (2018). https://xlab.tencent.com/special/spectre/spectre_check.html

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[36] Jonathan Levin. 2012. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons.

[37] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices.. In *USENIX Security Symposium*. 549–564.

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

[39] Google LLC. 2018. Google Chrome. (2018). https://google.com/chrome

[40] H. J. Lu. 2018. [PATCH 0/5] x86: CVE-2017-5715, aka Spectre. (2018). https://gcc.gnu.org/ml/gcc-patches/2018-01/msg00422.html

[41] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* 2, 1 (2018), 33–50.

[42] Clémentine Maurice and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, Vol. 10379. Springer, 161.

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

[44] Salvador Palanca, Stephen A Fischer, and Subramaniam Maiyuran. 2003. CLFLUSH micro-architectural implementation method and system. (April 8 2003). US Patent 6,546,462.

[45] Andrew Pardoe. 2018. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2018). https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[46] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*. Springer, 138–154.

[47] Yutaka Sasaki et al. 2007. The truth of the F-measure. *Teach Tutor mater* 1, 5 (2007), 1–5.

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

[49] Nikolay A Simakov, Martins D Innus, Matthew D Jones, Joseph P White, Steven M Gallo, Robert L DeLeon, and Thomas R Furlani. 2018. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. *arXiv preprint arXiv:1801.04329* (2018).

[50] Igor Sysoev. 2018. nginx. (2018). https://nginx.org

[51] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. 2014. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 109–129.

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

[53] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. (2018). https://support.google.com/faqs/answer/7625886

[54] Ganesh Venkitachalam and Michael Cohen. 2009. Transparent page sharing on commodity operating systems. (March 3 2009). US Patent 7,500,048.

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

[56] Amos Waterland. 2018. stress. (2018). https://people.seas.harvard.edu/~apw/stress/

[57] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*. 719–732.
[58] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 118–140.
[59] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. 2011. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy*. IEEE, 313–328.
[60] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 990–1003.