applied sciences

MDPI

Article

Real-Time Detection for Cache Side Channel Attack using Performance Counter Monitor

Jonghyeon Cho [1], Taehun Kim [2], Soojin Kim [2], Miok Im [2], Taehyun Kim [2] and Youngjoo Shin [2,*]

[1] Department of Computer Engineering, Kwangwoon University, Seoul 01897, Korea; whwhdgus94@naver.com

[2] School of Computer and Information Engineering, Kwangwoon University, Seoul 01897, Korea; taehunpb.kim@gmail.com (T.K.); kipper152@naver.com (S.K.); sjsqkqh33@naver.com (M.I.); taehyun9203@gmail.com (T.K.)

* Correspondence: yjshin@kw.ac.kr

Received: 18 December 2019; Accepted: 31 January 2020; Published: 3 February 2020

Abstract: Cache side channel attacks extract secret information by monitoring the cache behavior of a victim. Normally, this attack targets an L3 cache, which is shared between a spy and a victim. Hence, a spy can obtain secret information without alerting the victim. To resist this attack, many detection techniques have been proposed. However, these approaches have limitations as they do not operate in real time. This article proposes a real-time detection method against cache side channel attacks. The proposed technique performs the detection of cache side channel attacks immediately after observing a variation of the CPU counters. For this, Intel PCM (Performance Counter Monitor) and machine learning algorithms are used to measure the value of the CPU counters. Throughout the experiment, several PCM counters recorded changes during the attack. From these observations, a detecting program was implemented by using these counters. The experimental results show that the proposed detection technique displays good performance for real-time detection in various environments.

Keywords: cache side channel attack; Flush + Reload; Prime + Probe; Flush + Flush; Performance Counter Monitor

1. Introduction

Cache side channel attacks have become a major security threat for PC and cloud environments. This attack exploits the measured access time for the shared cache memory and then extracts sensitive information from the victim. There are a variety of well-known cache side channel attacks such as Flush+Reload [1], Flush+Flush [2], and Prime+Probe [3].

In Flush+Reload and Flush+Flush attacks, a spy infers the cache usage of a victim by measuring the access time of the cache line that is shared with the victim. On the other hand, the Prime+Probe attack targets a cache set that is shared with the victim to infer the cache usage. Since all of those attacks target the L3 cache, it is stealthy and unnoticeable to the victim. These attacks are also used to recover secret data from transient execution attacks such as Meltdown [4], Spectre [5], Foreshadow [6], and ZombieLoad [7]. To mitigate these transient execution attacks, CPU vendors have proposed several countermeasures that can significantly affect the performance. However, if it is possible to detect the cache side channel attack, it can provide a fundamental countermeasure against transient execution attacks. Therefore, it is necessary to study the detection of cache side channel attacks.

There are several techniques that can detect and block cache side channel attacks. While a spy executes an attack, these techniques measure the cache miss by using hardware performance counters while attempting to detect the cache side channel attacks based on the cache measurements. However,

these techniques have limitations since they are unable to detect attacks in real time and they depend on the cache miss.

To sum up, the previous detection methods have two limitations: (1) inability to detect attacks in real time and (2) lacking the detection of stealthy side channel attacks such as Flush+Flush, which incur no cache miss. In order to deploy the detection method in practice, we need to overcome these limitations.

In this article, we propose a novel technique to detect cache side channel attacks. The detection techniques are constructed upon classification-based machine learning algorithms and the Intel PCM (Performance Counter Monitor) [8]. This allows us to detect cache side channel attacks in real time without depending on the cache miss. The Intel PCM enables the measurement of the cache state during the attack in real-time. Machine learning algorithms detect anomalies based on the measured cache state. The experimental results show that the proposed technique successfully detects in real time all the cache side channel attacks including Flush+Flush attack with an average accuracy of 95%. This result proves that the proposed detection technique can be used as a useful tool to protect the security in a PC or cloud environment.

Contribution: This article has the following contributions. First, we propose a novel method that enables real-time detection for all kinds of cache side channel attacks. Second, we identify new available performance counters in the Intel PCM through extensive experiments and analysis, which allows us to detect stealth attacks. Third, we evaluate the effectiveness of the proposed detection method by conducting experiments in various execution environments including the virtualized environment and the single OS environment.

Outline: In Section 2, we present related work. In Section 3, we describe some background information about cache side channel attacks, Intel PCM, and the softmax classification algorithm. In Section 4, the experimental results are presented to identify new useful CPU counters for detection in the Intel PCM. In Section 5, we propose a detection method of the cache side channel attacks based on a machine learning algorithm. In Section 6, we evaluate the effectiveness of the detection program. Finally, we conclude by summarizing our work in Section 6.

2. Related Work

There were several previous works regarding the detection of cache side channel attacks. Chiappetta et al. proposed three methods to detect the Flush+Reload attack by using the *perf* command and machine learning techniques [9]. The first method was based on correlation and the other two use machine learning techniques. The method proposed in the article had the advantage of being able to detect and prevent attacks in a relatively short time. However, they only used a cache miss counter; hence they could not detect the Flush + Flush attacks, which did not affect the cache misses [2]. Mushtaq et al. [10] used 12 hardware events to train 12 machine learning models. They could detect the Prime+Probe attack using real-time data by selecting the best four models from 12 machine learning models. Based on the relatively large number of 12 hardware events, fast and accurate attack detection was possible with a low overhead. However, this had the same limitation as Chiappetta et al. [9] since they were unable to detect the Flush+Flush attack. Mohammad-Mahdi et al. [11] proposed an approach to detect the cross-Virtual Machine (VM) cache-based side channel attacks using the hardware granular information provided by the Hardware Performance Counters (HPCs) and the Intel Cache Monitoring Technology (CMT) according to the Gaussian anomaly detection method. This study had the advantage of a high detection rate because the performance overhead was 2% for the computing platform. However, their method determined which attacks were based on cache misses that were affected by high false positive alarms. Therefore, our technique, which does not rely on cache misses, would likely be more appropriate. Gulmezoglu et al. [12] proposed a method for detecting multiple attacks (e.g., Flush+Reload, Prime+Probe, Rowhammer [13], Spectre, Meltdown, Zombieload) using the Intel PCM and deep learning. In this investigation, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) algorithms, which are deep leaning models, were trained

with performance counters, such as L1_INST_MISS, L1_INST_HIT, and LLC_MISS, to predict the next counter value. The attack was detected through a difference between the predicted counter value and the actual counter value. This method differs from the proposed method, which is capable of detecting the attacks by predicting the counter values. Unlike previous methods, it has the advantage of detecting multiple attacks. However, the reliability of the proposed method is higher since it trains machine learning models with five performance counters in comparison to the method proposed by Gulmezoglu et al. [12] Furthermore, this method requires much computation and many data for training, which uses deep learning. However, the proposed machine learning technique from this study is more suitable for real-time detection because it requires less computation and training data. Table 1 summarizes the comparison to other related works.

Table 1. Comparison to related works.

Method	Tool	Performance Counters	Detected Attacks	Can Detect Stealth Attack?
Chiappetta et al. [9]	Linux perf	LLC_MISS	Flush+Reload Prime+Probe	No
Mushtaq et al. [10]	Intel CMT	L1_MISS LLC_MISS	Flush+Reload Prime+Probe	No
Mohammad-Mahdi et al. [11]	Intel CMT	L1_MISS LLC_MISS	Flush+Reload Prime+Probe	No
Gulmezoglu et al. [12]	Intel PCM	L1_INST_MISS L1_INST_HIT LLC_MISS	Flush+Reload Prime+Probe	No
The proposed method	Intel PCM	IPC L1_MISS L2_MISS LLC_MISS RETIRED_BRANCH	Flush+Reload Flush+Flush Prime+Probe	Yes

3. Background

3.1. Cache Side Channel Attacks

3.1.1. Flush+Reload Attack

Flush+Reload [1] is a cache side channel attack that aims at L3 or the Last Level Cache (LLC). For a modern Intel x86 multi-core architecture, each core has a separate L1 and L2 cache and an LLC shared by all of the cores. Owing to the cache inclusive property, the LLC can possibly allow information to be leaked through the cache to any malicious users or Virtual Machines (VMs). A virtualized environment, such as cloud services, sometimes uses memory sharing (i.e., content-based sharing, memory deduplication) to reduce the duplicated memory usage. In a general environment without using virtualization, the operating systems use content-aware sharing to share shared libraries and a shared code section. Despite the benefits of memory sharing, it can be used as a target of the cache side channel by sharing a physical page with a spy. By monitoring the shared LLC line, a spy knows whether the victim accesses the sensitive data in sharing pages. The Flush+Reload attack consists of three steps as follows. 1. FLUSH: A spy flushes the specific shared cache line by using a clflush instruction. 2. IDLE: Then, the spy waits for a predetermined amount of time while the victim executes sensitive operations. 3. RELOAD: The spy reloads the cache line from the shared memory. If it takes too long to reload the cache line, it means that the victim did not access the sensitive data in the shared page. However, if the reload time is short, the shared cache line is filled with the victim's data, which means that the victim accesses sensitive data. By using this reload time difference, the spy is able to infer the victim's access pattern for some sensitive data from the LLC cache line. Since Flush+Reload

has the highest resolution of cache side channel attacks, it is used not only to extract the secret key of various encryption algorithms, but it also obtains the victim's keystroke information. Because the attack uses the clflush instruction for the flush specific cache line iteratively, an increase in the L1, L2, and LLC miss occurs whenever the victim accesses the cache lines.

3.1.2. Flush+Flush Attack

Similar to Flush+Reload, Flush+Flush targets the LLC line shared between the victim and the spy. This attack uses the time difference between two clflush instructions, rather than the time difference between the cache hits and misses for the reload after the clflush instruction in the Flush+Reload attack. Because the clflush instruction, which is irrelevant to the memory access, is faster than the memory access instruction, it does not cause any other L1, L2, LLC miss, or hit. Therefore, the Flush+Flush attack would ideally be a fast and stealthy cache attack. However, the time difference between the two clflush instructions, with and without data in the cache line, is less noticeable than the difference in the reload time for the Flush+Reload attack. Therefore, Flush+Flush has less accuracy than Flush+Reload [2]. The Flush+Flush attack consists of three steps as follows: FLUSH, IDLE, and FLUSH. This attack has the same steps as Flush+Reload except for the final FLUSH step. In the final FLUSH step, the spy measures the execution time of the clflush instruction. If the execution time of the clflush instruction is measured to be too long, this implies that the victim accessed the probing cache line or the sensitive data. If the execution time of the clflush instruction is short, it means the victim did not access the probing cache line.

3.1.3. Prime+Probe Attack

Unlike Flush+Reload and Flush+Flush, which target the shared cache line, the Prime+Probe attack aims at the Last Level Cache (LLC) set. Because the LLC can contain the shared memory between the cores, the spy does not need to prepare the shared memory. Therefore, the Prime+Probe attack can be applied more broadly than the other attacks. As the Prime+Probe targets the cache set, it has a lower resolution than the Flush+Reload and Flush+Flush, which target the shared cache line. To make the sharing cache set, the spy makes an eviction_set that shares a cache set between the victim and the spy for this technique. By probing all of the lines for the eviction_set, the spy knows whether the victim accesses the sensitive data. The attack consists of three steps as follows. 1. PRIME: A spy fills the cache sets with data. 2. IDLE: The spy waits for a predetermined amount of time while the victim executes the sensitive operations. 3. PROBE: The probe caches the sets with prepared data. If the probing time is measured to be too long, or if the eviction_set is changed, it implies that the victim accessed the cache set while evicting some cache lines of the cache sets.

3.2. Performance Counter Monitor

The Performance Counter Monitor (PCM) is a tool that allows users to monitor the performance counter values of the CPU core and uncore (i.e., read and written bytes from the memory controller). It is similar to the Performance Application Programming Interface (PAPI). PCM helps the users to monitor the internal counter (e.g., instruction per cycle, L1, L2, and L3 cache miss) change rate of each CPU in real time. PCMs typically have *perf* and Intel PCM and present their results using special-purpose registers inside the CPU. *perf* is a command line tool to analyze the performance in Linux. It can be visualized and aggregated through the Hardware Performance Counter (HPC). Chiappetta et al. [9] proposed a detection method for cache side channel attacks using *perf*. The Intel PCM supports the Windows environment other than the Linux environment, and it is similar to *perf*. However, Intel PCM can be run as a binary file by performing a compile. Unlike *perf*, Intel PCM has a CSV option that prints several counter values of the CPUs to a CSV file for a set time. Intel PCM can be used to distinguish the characteristics of cache side channel attacks through a change of counters. This is suitable for monitoring the experiment that is carried out in this article.

3.3. Softmax Classification

Softmax classification is used for multi-label classification, which classifies given data into three or more labels. This contrasts with binary classification, which separates the given data into two labels. The softmax function converts the predicted value of each label for an input to a probability value between zero and one. The sum of the probabilities that belong to each label should be equal to one. In particular, p_i is the probability that the input belongs to each label; the softmax function defines p_i as the following Equation (1) [14].

$$p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \text{ for } i = 1, 2, \ldots, k. \quad (1)$$

As the probability values arrive closer to one through the softmax function, the higher the probability belonging to the corresponding label. Therefore, through one-hot encoding, where only one is true for two or more inputs and all others are false, the probability of having the largest value is one, and the rest are zero. Softmax classification is applied to the cross-entropy loss up to the softmax function. The cross-entropy is a function that reduces the error between the actual and predicted values. The actual value is either zero or one through one-hot encoding of the predicted value; hence, the log value gives a zero or infinite value. If the predicted value is different from the actual value, the cost will have a very large value. Based on this, the machine learning model is trained so that the loss function has a minimum value.

4. Identifying New Performance Counters

This section describes the experiment to obtain useful CPU counters for detecting cache side channel attacks such as Flush+Reload, Flush+Flush, and Prime+Probe. In our previous study, it was only shown that the Flush+Reload attack incurred a significant cache miss rate in the PCM [15].

The experiments in this study were performed on an Asus X99-E WS server computer with Ubuntu 16.04 LTS, an Intel Xeon® E5-2620 v4 processor, and a 32GB DDR3 memory as the victim. As a spy, a MacBook Air with Mojave OS and an Intel Core i5-5250U 1.6GHz processor and 8GB DDR3 memory was used. Attacks proceeded via SSH remote access. Three cache side channel attacks were implemented by using Mastik [16], a cache side channel framework. Mastik makes it easy to execute various types of cache side channel attacks with a simple setup and compilation. A PCM tool that was provided by Intel was used as an open source [8]. Intel PCM offers many options; thus, CSV file options were used to extract the counter values, which created the graphs to easily see changes in the counter values due to the attacks.

Since these cache side channel attacks can be easily mounted in various environments, it is necessary to observe the PCM counter value in various situations. Thus, experiments were performed with two scenarios. First, in a normal scenario, the attacks were tested without background applications. In the second scenario, the attacks were tested with several applications such as a video player, an office program (e.g., Libre Office Calc), and a web browser (e.g., Firefox) running in the background. During the experiments, all of the cache side channel attacks were executed for 15 s each in a normal scenario and the running application scenario. Prior to mounting the attacks, 15 s of wait time were provided for each scenario to clearly identify the change in the PCM counters.

Through this experiment, five counters were discovered that significantly changed due to the three attacks. These counters include the IPC (Instruction Per Cycle), L3 cache miss, L2 cache miss, L1 cache miss, and the speculative and retired branch counters. Each attack had a slightly different effect on the counters, except the IPC (average number of instructions per cycle) value showed identical changes to all of the attacks.

Figure 1 shows the changes of the IPC value when the attack was executed in each scenario in the experiment. In both scenarios, the IPC value was greatly reduced when the attacks were executed. The reason for greatly changing the IPC value was that the memory access had different characteristics

for the three attacks. The Flush+Reload attack resulted in frequent main memory accesses in the process of spy flushing and reloading the cache line, which consumed many cycles. The Prime+Probe attack occurred mostly for main memory access at other logical cores that shared the cache set in the process that spy filled and probed its own data. Unlike the other two attacks, the Flush+Flush attack did not have access to the main memory; however, it took many cycles to continuously flush the cache line [2].

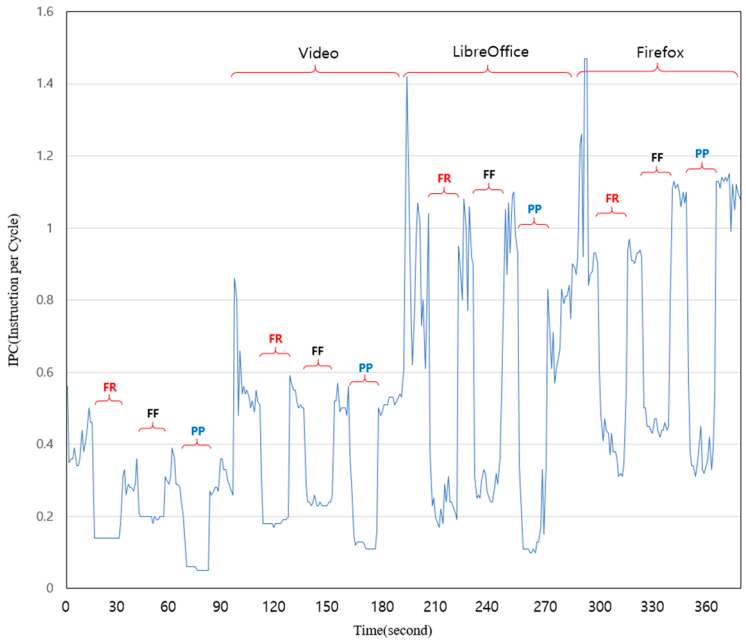

Figure 1. The changed values of the IPC counter when the three attacks were executed in two scenarios.

Figure 2 shows the change in the L1, L2, and L3 cache misses when the attack was executed. In both scenarios, three cache misses were significantly increased when the Flush+Reload and Prime+Probe attacks were performed. However, the increment in the cache misses counter value was different for both attacks. In the case of the Flush+Reload attack, it was observed that the three cache misses were increased by approximately two million. On the other hand, in the case of the Prime+Probe attack, the L1 and L2 cache misses were increased by approximately 50 million; however, the L3 cache miss did not increase. It was observed that there was no change when the Flush+Flush attack was performed. The reasons for these changes were considered. For the case of Flush+Reload, it caused the cache misses to be increased when the spy continued to flush and reload the cache line while waiting for the victim's access. For the Prime+Probe attack, the cache miss did not occur in L3 because the spy filled its own data in the cache set and probed while waiting for the victim to access it. However, L1 and L2 were shared by other logical cores that used hyper-threading. As a result, the cache misses were significantly increased.

Figure 3 shows the change in the value of the speculative and retired branch counter when the three attacks were executed in each scenario of the experiment. The CPU executed the speculative execution that pre-performed some tasks in order to improve the performance in executing instructions. The speculative and retired branch counter counted the retired branches that were not taken through the speculative execution. The counter was used to carry out the experiment. This counter increased greatly when a program was run and when the program contained many loops. Furthermore, the counter value was observed to rise suddenly and remain steady when the three attacks were executed. It was hypothesized that this was due to a feature of the three attacks that constantly accessed the cache line or set.

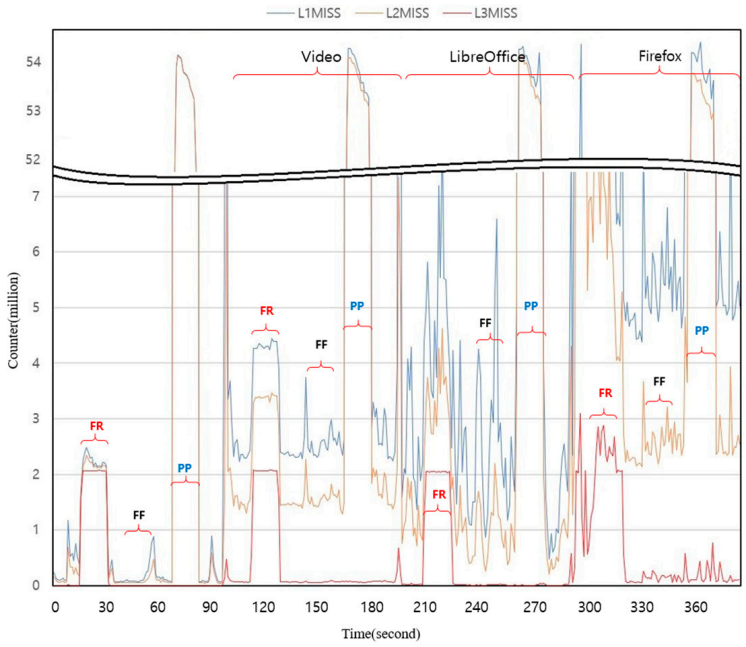

Figure 2. The changed values of the L1, L2, and L3 cache miss counter when the three attacks were executed in two scenarios.

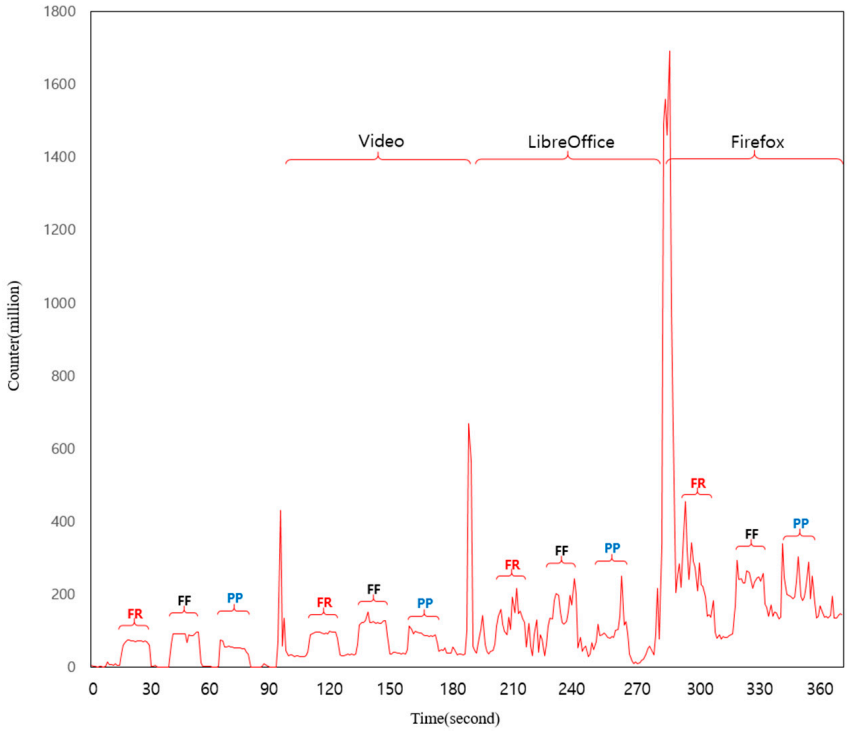

Figure 3. The changed values of the speculative and retired branch counter when the three attacks were executed for two scenarios.

The experimental results showed that the cache side channel attacks could be distinguished using several counters. In all three attacks, the IPC value substantially decreased. Furthermore, the speculative and retired values increased when each attack was executed. In this situation, if the cache miss counters were significantly increased by more than 50 million, this was determined to be a Prime+Probe attack. If the cache miss counters were increased by more than two million, then this was a Flush+Reload attack. If there was no change in the cache miss counter, this was a Flush+Flush attack.

5. Real-Time Detection Using PCM

5.1. Overview and Threat Model

In this section, we give an overview of the proposed detection method against cache side channel attacks, as well as its threat model. Our proposed method basically tried to detect the attack based on abnormal behavior of processors. For this, the method internally utilized an Intel PCM tool for the detection. The PCM tool collected a variety of hardware performance counters associated with the processor events in real time and provided the results through a file with CSV format. Our detection program repeatedly read relevant performance counters from the CSV file and checked whether some abnormal events (i.e., cache side channel attacks) happened in the processor. If so, the program further predicted what kind of attack was in progress based on classification-based machine learning algorithm. Since the detection program directly read some counters from the processor status registers, we required the program to run in privileged mode (i.e., root mode).

We supposed that in practice, the detection program would be deployed and utilized in two possible execution environments: (1) a virtualized environment and (2) a single OS environment. In this article, we followed the threat models of other related studies [4,5,12,14–16] for the execution environments. We describe each environment in detail, as well as its threat model in the following:

- Virtualized environment: The detection program ran in one of the Virtual Machines (VMs) on the host (see Figure 4a). The attacker process was located on the same host, but ran in another VM. In our threat model, we assumed that the attacker owned its VM and had a root privilege to a guest OS in the VM. Despite its privileged access to its own VM, however, the attacker could not interfere with the detector program by conducting sabotage on the execution file or CSV files as the hypervisor logically separated these two VMs from each other. Cloud computing was one of the possible scenarios for the virtualized environment.
- Single OS environment: The attacker and the detector program ran in the same operating system on the host (see Figure 4b). In our threat model, we assumed that the attacker only had the user-level access to the OS in this environment. Owing to the privilege-based access control of the operating system, we could restrict access permission to any relevant resources such as CSV files so that only a root process including the detection program could access them. By doing so, the attacker still could not conduct sabotage on the executables and CSV files of the detector program.

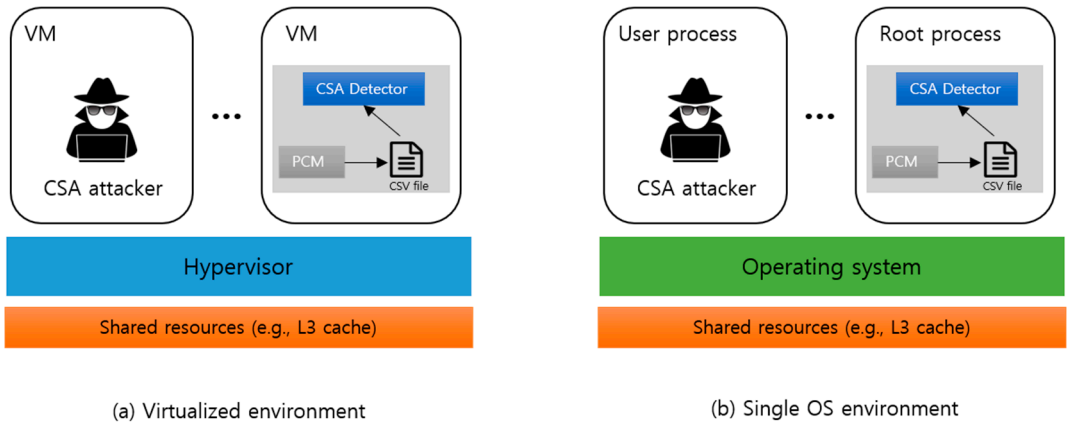

Figure 4. Architecture of the proposed detection method and its execution environment.

5.2. The proposed Detection Method

Now, we describe the proposed detection method to cache side channel attacks in detail. Intel provides a PCM tool that extracts a wide variety of counters value and stores the results with a CSV file format. Although the PCM tool is easy to use, it has several unnecessary counters and consumes many

system resources [15]. Therefore, we modified the PCM tool to make it faster and more lightweight so that was suitable for real-time detection. The modified tool could extract the counter value with the given umask and event by using less CPU than the original PCM tool. In our experimental environment, we used an Intel Xeon® E5-2620 v4 processor with the Haswell-EP microarchitecture. The software developer's manual for Intel processors describes in detail all the performance counters available for users including their umasks and event numbers. Among them, we identified four performance counters suitable to our side channel detection method. Table 2 shows the counter names, umasks, and events of the four counter values that were used in our environment. In addition to these counters, we also used a counter for IPC, which is internally provided by the PCM-core, for our detection algorithm. Note that all the identified counters were not specific to the Xeon E5-2620 v4 processor, but were applicable to any Intel processors.

Table 2. Performance counters used in the proposed detection method.

Counter Name	Umask	Event
MEM_LOAD_UOPS_RETIRED.L1_MISS	0 × 08	0 × d1
MEM_LOAD_UOPS_RETIRED.L2_MISS	0 × 10	0 × d1
MEM_LOAD_UOPS_RETIRED.L3_MISS	0 × 20	0 × d1
BR_INST_EXEC.ALL_BRANCHES	0 × ff	0 × 88

In order to detect the attack by using the identified performance counter, we used a machine learning technique. In particular, we used a softmax classification algorithm for the proposed method. The softmax classification allowed us to detect the attack and further to classify it according to several known attacks. For the purpose of classification, we labelled each known attack to a specific constant number. More specifically, we labelled the Flush+Reload attack, the Flush+Flush attack, and the Prime+Probe attack A1, A2, and A3, respectively. For a normal situation without any attacks, we labelled it A0.

Figure 5 illustrates the overall architecture of our machine learning model for the attack detection. The model consisted of a single-layer perceptron without hidden layer with five units in the input layer and one unit in the output layer. Units in the input layer (i.e., x_1 ~x_5) corresponded to the collected values from performance counters listed in Table 2 along with the IPC counter. The output unit y_1 referred to one of the labels A0~A3 based on the classification.

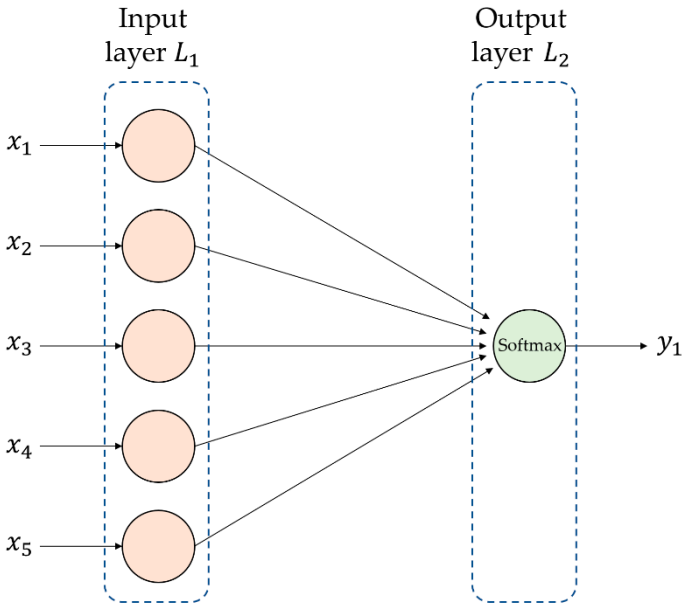

Figure 5. The machine learning model used in the proposed detection method.

We performed training by using the performance counters obtained from our PCM tool. Since the performance counters were likely to be influenced by other benign applications running on the host, we obtained the training data not only from the normal condition (i.e., no applications running), but also from certain conditions where various applications were running concurrently such as web browsers and media players. After training, we obtained 3500 samples of performance counters from our PCM tool. The training data were gathered according to various scenarios. That is, we gathered the data with or without mounting side channel attacks while also simultaneously running several background applications. As a result, we obtained the softmax classification model, which was trained with specified labels.

We implemented the softmax classification model by using TensorFlow, an open source-based framework for machine learning algorithms provided by Google. We specifically configured the TensorFlow parameters as follows: the learning rate was set to 0.1, the epochs to 100, and the batch size to 100. Moreover, we used the Adam optimization [17] technique to improve detection accuracy by minimizing the cost. The training model showed about a 99.54% accuracy for the training cases. The prediction model we configured using softmax classification detected and classified the three attacks and the normal condition with input values, that is IPC and four counter values. The output was one of the labels designated by softmax classification.

Based on the classification model, we implemented a detection program to defend against three cache side channel attacks. Since the model was built with TensorFlow, we implemented the detection program by using Python for compatibility. Algorithm 1 presents the pseudo-code of the overall execution process of the detection program. Once the program was executed, it first forked the process by using a fork function (Line 1 of the algorithm). As a result, child and parent processes could run simultaneously. In Line 3, the child process was invoked and continued to execute our modified PCM. The PCM tool was provided with an option of using CSV file format. On the other hand, the parent process performed appropriate initialization process and then began execution by loading the machine learning model that we built (Line 5). The PCM tool performed monitoring and continuously output the collected counter values including the IPC, L1 miss, L2 miss, L3 miss, and the speculative and retired counters to the CSV file. Then, the program loaded the values in the CSV file and ran the model with the collected values as inputs to predict the current situation (Line 8). If the current situation was under attack, the specified label (A1: Flush+Reload, A2: Flush+Flush, and A3: Prime+Probe) was used to display the result of the attack (Lines 9-14). Finally, the CSV file was managed by flushing it. This prevented it from being slowed down when reading the CSV file because it was accumulating data.

Algorithm 1 Pseudo-code of the detection program

```
1  P = fork()
2  if P is child process
3      Run PCM with an option of using CSV format
4  else
5      model = Load machine learning model file and initialize.
6      While(always) do
7          I = open and read CSV file
8          Result = model ← I
9          if Result is A1
10             Detect Flush+Reload
11         else if Result is A2
12             Detect Flush+Flush
13         else if Result is A3
14             Detect Prime+Probe
15         flush CSV file
```

6. Evaluation

In this section, we conduct several experiments and validate the effectiveness of the proposed detection technique. Specifically, we evaluate our detection method by testing whether the machine learning model was able to detect cache side channel attacks in several environments. In order to evaluate the performance of the detection program, we set up two execution environments as described in Section 5.1. In the virtualized environment, we used KVM (Kernel-based Virtual Machine) as a hypervisor and Linux Ubuntu 18.04 LTS (64 bits) as the guest operating systems. In the single OS environment, we set up Ubuntu 18.04 LTS as a host operating system. To validate the wide applicability to various processor models, we performed experiments on six hosts in total with different processor models including Intel Xeon and Core.

We also needed to evaluate the detection performance in practical environments where the detection program ran concurrently with other benign applications. In our experiment, we simulated the activities of benign applications by using a *stress-ng* tool. The stress-ng incurs stress on various system components such as CPU, memory, and I/O. As our program attempted to detect malicious behavior based on cache activities, we used the stress-ng with an option "-c" that incurred stress on the CPU cache.

The experimental results are presented in Tables 3 and 4, for the virtualized environment and the single OS environment, respectively. The detection rate was measured by counting the number of success in detection within five seconds after mounting the cache side channel attack. In the experiment, we set the time limit of detection to five seconds. This was based on the results of previous work that studied the minimal amount of time to recover target's secret through cache side channel attacks [18]. The time in the tables refers to the elapsed time of successfully detecting the attacks.

The detection program was first tested on the Intel Xeon® E5-2620 v4, which was used to build the model and run the experiments. The *htop* tool was used to check the CPU usage in real time of the detection program and to evaluate the program with several applications. According to the experimental results, high detection rates and a low CPU usage were observed in the E5-2620 v4 to detect all of the attacks. In order to verify the performance of the detection program for the different CPUs, we had to learn whether the umask and the event that were used in Table 2 were the same as in the other microarchitectures. The event and umask of the Broadwell, Kaby Lake, and Coffee Lake microarchitectures are found in Intel's Software Developer Manuals. In addition, the values were checked to make sure they were the same as Table 2.

Table 3. Experiment results in the virtualized environment.

CPU (Codename)	Normal (No Stress)			Cache Stress (stress-ng)		
	Detection Rate (%)	Time (second)	CPU Usage (%)	Detection Rate (%)	Time (second)	CPU Usage (%)
Intel Xeon® E5-2620 v4 2.10 GHz (Broadwell)	98.4%	1.9	0.6%	92.5%	2.1	0.9%
Intel Xeon® E3-1275 v6 3.80 GHz (Kaby Lake)	97.1%	1.9	0.8%	91.9%	2.2	0.9%
Intel Core™ i5-7400 3.00 GHz (Kaby Lake)	96.2%	1.8	0.7%	92.9%	2.3	0.8%
Intel Core™ i7-7700 3.60 GHz (Kaby Lake)	98.7%	1.9	0.8%	92.3%	2.2	1.1%
Intel Core™ i7-9700 3.60 GHz (Coffee Lake)	94.5%	1.7	0.9%	92.9%	2.4	1.1%
Intel Core™ i5-5250U 1.6 GHz (Broadwell)	97%	1.8	0.9%	92.8%	2.2	0.8%

Table 4. Experiment results in the single OS environment.

CPU (Codename)	Normal (No Stress)			Cache Stress (Stress-ng)		
	Detection Rate (%)	Time (second)	CPU Usage (%)	Detection Rate (%)	Time (second)	CPU Usage (%)
Intel Xeon® E5-2620 v4 2.10 GHz (Broadwell)	100%	1.5	0.7%	95%	1.8	0.9%
Intel Xeon® E3-1275 v6 3.80 GHz (Kaby Lake)	99.8%	1.6	0.7%	94.1%	1.9	0.9%
Intel Core™ i5-7400 3.00 GHz (Kaby Lake)	99.6%	1.6	0.6%	94.2%	1.9	0.8%
Intel Core™ i7-7700 3.60 GHz (Kaby Lake)	99.2%	1.7	0.7%	94.3%	1.9	0.9%
Intel Core™ i7-9700 3.60 GHz (Coffee Lake)	99.5%	1.5	0.7%	95.6%	1.8	0.9%
Intel Core™ i5-5250U 1.6 GHz (Broadwell)	100%	1.5	0.6%	95.4%	1.8	0.9%

In order to verify the exact attack only, programs were found that could change the counter values used in the detection program, but were not attacked. The *stress* command can overload the Linux environment, which can assume a situation where many users use the environment to generate many cache misses. Therefore, it was assumed that the attacks could be applied to the server computer or the cloud environment that many users used. We also considered how to increase the speculative and retired branch counter. Hence, a loop code was used that repeated the rand function infinitely. This is because many loops resulted in a speculative execution and retirement when executing the program. The detection program was evaluated using a basic machine learning model. However, in the loop code and *stress* environment, all three attacks were detected as an attack; however, each attack was not distinguished. Therefore, additional models were created to distinguish all three attacks in each experimental environment. The added model used 30 s of the three attack data and was normal for each environment, which would give the appropriate label to the model. In the loop code environment, the three attacks could be detected and distinguished with an average 95% probability in all six CPU environments using the model. However, in environments that contained stress, the machine running model could not simultaneously detect and distinguish the three attacks due to different overloaded systems for each CPU. Therefore, 30 s of separate training were conducted in stressful environments for all six CPUs. This was the same as the previous training sessions, and the detection program was able to detect all of the attacks with a high probability in an overloaded environment.

This study showed through the evaluation process that the proposed detection programs had high detection rates. Furthermore, the problem of misdetection rates was solved using a short training process, which gave additional data in different environments. Therefore, these training processes could be automated by writing a program that would create an optimal model for the current environment in about two minutes.

7. Conclusions and Future Work

This article proposed a runtime detection technique for the cache side channel attacks by using the Intel PCM counters and machine learning algorithms. We showed that the proposed detection method was effective at detecting cache side channel attacks by using newly identified hardware counters. These counters included the IPC (Instruction Per Cycle), L3 cache miss, L2 cache miss, L1 cache miss, and the speculative and retired branch counters. By using hardware performance counters, we trained the machine learning model to detect attacks targeting the L3 cache. This study demonstrated that the proposed technique could distinguish all kinds of cache side channel attacks

with more than 90% probability in real time. Besides, we showed that a trained model could detect many environments based on rigorous evaluation. By training the machine learning model through a short training process, our method was able to detect all kinds of cache side channel attacks, even in environments where misdetection would occur. This study used other factors, including cache misses, to detect the attack process. Therefore, we expect that our technique may be able to successfully detect any unidentified cache-based side channel attacks. Although our evaluation showed that the proposed method was feasible in an experimental setting, there are some remaining issues to be solved for practical deployment. Concerning security, the proposed method will be vulnerable if an attacker has privileged access to the critical resources (e.g., a CSV file) for running the detection program. Furthermore, from the performance perspective, the latency in reading and parsing the CSV file will be intolerable in the practical system where real-time detection is necessary. In our future work, we will address those remaining problems and continue our study to improve the proposed method.

Author Contributions: J.C. mainly wrote this article. T.K. (Taehun Kim), S.K., M.I., and T.K. (Taehyun Kim) contributed to this work by conducting experiments and performance analysis. Y.S. revised this article and contributed to the performance analysis. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MIST (Ministry of Science and ICT) under the National Program for Excellence in SW (2017-0-00096) supervised by the IITP (Institute for Information & communications Technology Planning & Evaluation) and was supported by the IITP grant funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yarom, Y.; Falkner, K. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014.
2. Gruss, D.; Maurice, C.; Wagner, K.; MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In Proceedings of the DIMVA'16, Donostia-San Sebastián, Spain, 7–8 July 2016.
3. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-Level Cache Side-Channel Attacks are Practical, Security Privacy. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015.
4. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 15–17 August 2018.
5. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre attacks: Exploiting speculative execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2019.
6. Bulck, J.V.; Inkin, M.; Eisse, O.; Genkin, D.; Kasikci, B.; Piessens, F.; Silberstein, M.; Wenisch, T.F.; Yarom, Y.; Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 15–17 August 2018.
7. Schwarz, M.; Lipp, M.; Moghimi, D.; Bulck, J.V.; Stecklina, J.; Prescher, T.; Gruss, D. ZombieLoad: Cross-Privilege Boundary Data Sampling. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; ACM: New York, NY, USA, 2019.
8. Willhalm, T.; Dementiev, R. Intel®Performance Counter Monitor—A Better Way to Measure CPU Utilization. Available online: https://software.intel.com/en-us/articles/intel-performance-counter-monitor (accessed on 16 August 2012).
9. Chiappetta, M.; Savas, E.; Yilmaz, C. Real time detection of cache-based side channel attacks using hardware performance counters. *Appl. Soft Comput.* **2016**, *49*, 1162–1174. [CrossRef]
10. Mushtaq, M.; Akram, A.; Muhammad, K.B.; Rao, N.B.R.; Lapotre, V.; Gogniat, G. Run-time Detection of Prime+Probe Side-Channel Attack on AES Encryption Algorithm. In Proceedings of the 2018 Global Information Infrastructure and Networking Symposium, Thessaloniki, Greece, 23–25 October 2018.

11. Mohammad-Mahdi, B.; Thibaut, S.; Marc, L.; Sudholt, M.; Menaud, J. Cache-based side channel attacks detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In Proceedings of the 2018 Third International Conference on Fog and Mobile Edge Computing, Barcelona, Spain, 23–26 April 2018.
12. Gulmezoglu, B.; Moghimi, A.; Eisenbarth, T.; Sunar, B. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv* **2019**, arXiv:1907.03651.
13. Gruss, D.; Maurice, C.; Mangard, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 300–321.
14. Memisevic, R.; Zach, C.; Pollefeys, M.; Hinton, G. Gated Softmax Classification. In Proceedings of the Advances in Neural Information Processing Systems 23, Vancouber, BC, Canada, 3 June 2010.
15. Cho, J.H.; Kim, T.H.; Shin, Y.J. Real-time detection on FLUSH+RELOAD attack using Performance Counter Monitor. *KIPS Trans. Comput. Commun. Syst.* **2019**, *8*, 151–158.
16. Yarom, Y. Mastik: A Micro-Architectural Side-Channel Toolkit. Available online: https://cs.adelaide.edu.au/~{}yval/Mastik/ (accessed on 17 August 2016).
17. Kingma, D.; Ba, J. Adam: A method for stochastic optimization. In Proceedings of the International Conference on Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015.
18. Irazoqui, G.; Inci, M.S.; Eisenbarth, T.; Sunar, B. *Wait a Minute! A Fast, Cross-VM Attack on AES, Research in Attacks Intrusions and Defense*; Springer International Publishing: Berlin/Heidelberg, Germany, 2014; pp. 299–319.