Protecting Shared Virtualized Environments against Cache Side-channel Attacks

Abdullah Albalawi, Vassilios G. Vassilakis and Radu Calinescu
Department of Computer Science, University of York, U.K.

Keywords: Side-channel Attacks, Cache Attacks, Prime+Probe, Flush+Reload, Flush+Flush.

Abstract: We introduce a side-channel attack detection and protection method that combines dynamic and static analysis. The dynamic analysis uses Linux Perf to obtain readings from 13 hardware performance counters related to the shared cache. Based on these readings, the virtual machine (VM) behaviour is then classified into suspicious or benign using logistic regression classification. As a second step, the static analysis extracts the executable files from the disk image or the RAM image of the suspicious VM. It then checks whether these files contain operating codes for side-channel attacks. Based on this, the threat level of these files is determined using the SoftMax classification algorithm; we have four threat levels in total. After that, VMs that pose a threat to the shared environment are excluded. As a hypervisor, we employed KVM (Kernel-based Virtual Machine), and as guest operating systems, we utilized Linux Ubuntu 18.04.5 LTS (64bits). We then conducted experiments on several host machines, namely Ubuntu 18.04.5 LTS, Debian 10, and CentOS 8, with various processor models. The accuracy of detecting suspicious behaviour and classifying the threat level was recorded as 96%–99% with between 0.6%–25% CPU overheads for dynamic and static analysis.

1 INTRODUCTION

Cloud computing offers the advantage of sharing the computing power of cloud-based hardware and software resources. In doing so, cloud computing has become one of the widely adopted approaches to reduce costs for organizations, decrease administrative procedures for users, and optimise computing power utilization. However, the shared computing resources are vulnerable to exploitation by malicious users leading to confidentiality violations, such as cracking encryption keys and exposing sensitive data (Saxena et al., 2017). The malicious user can exploit the shared resources to perform cache side-channel attacks to expose the victim's sensitive information.

Although several mitigation approaches for cache attacks have been presented (Irazoqui et al., 2018; Bazm et al., 2018; Mushtaq et al., 2018; Cho et al., 2020; Chiappetta et al., 2016), a number of limitations exist. First, since it increases the execution time, these approaches do not involve precautionary actions when detecting malicious executable files, particularly when performing static analysis. Some of these solutions do not have a signal to start for performing static analysis, which is generally known to cause significant overload of the system. Additionally, most of the previous methods are required to enhance performance and detection results. Finally, none of the most well-known antivirus tools are able to detect the attacks we have analyzed (i.e., cache side-channel attacks) for protecting shared virtualized environments (Irazoqui et al., 2018).

In this paper, we introduce a new method that combines dynamic and static analysis to detect and defend against cache side-channel attacks. The dynamic analysis approach monitors the activities of virtual machines (VMs). It detects suspicious activities that indicate the presence of cache side-channel attacks by extracting readings from hardware performance counters using Linux *P*erf and classifying them using logistic regression to determine whether or not this is an attack.

If suspicious activity is detected for one of the VMs, this is considered the starting signal for operating the static analysis of the executable files of the suspicious VM. The VM's executable files are accessed, disassembled, and analysed to find out if they contain implicit characteristics and the operation codes (opcodes) of the attacks. The threat level of these files is then determined using a neural network classifier that uses the SoftMax algorithm for classification. Hence, this paper makes the following primary contributions:

1. We propose an approach for detecting and protecting shared virtualized environments against cache side-channel attacks using dynamic and

Albalawi, A., Vassilakis, V. and Calinescu, R.
Protecting Shared Virtualized Environments against Cache Side-channel Attacks.
DOI: 10.5220/0010897800003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 507-514
ISBN: 978-989-758-553-1; ISSN: 2184-4356

static analysis.

2. We describe the method and results of the mechanism design, implementation, and experimentation.
3. We evaluate our approach in various attacks scenarios in terms of detection efficiency and performance attributes.

The remainder of the paper is arranged as follows. Section 2 presents the background information on cache side-channel attacks. In Section 3, we state the problem. Section 4 illustrates the proposed detection and protection method. Section 5 describes the experimental investigations based on the proposed method. In Section 6, the evaluation of the implemented method is discussed. Section 7 presents a comparison of the method presented in this paper with related previous work. Finally, Section 8 provides the conclusion and directions for future work.

2 BACKGROUND

2.1 Cache Side-channel Attacks

In cache side-channel attacks, the attackers exploit timing information gathered by identifying the difference in time between obtaining data from the cache and memory to attack the shared cache in virtualized environments (Anwar et al., 2017).

Data can be obtained whether from the memory or the cache when the CPU searches for it. When data is retrieved from the cache, it takes very little time or CPU power to do it. However, let's say the data is not stored in the cache, which means the data will be recovered from the system's main memory, which will require more time and CPU power. The data obtained from the main memory will be temporarily stored in the cache to decrease the system degradation if the same cache line is required later. Thus, the attack process relies on the timing information between obtaining data from the cache and the main memory (Irazoqui et al., 2014).

Utilizing timing information of retrieving data on the system, the attackers execute cache attacks on the victim's VM in a shared virtualized environment, measuring the CPU cycles or the time taken for accessing the cache targeted locations using cache hits and cache misses measurements. This technique allows the attacker to breach the VM isolation, learn about the victim's activities on the shared cache.

The following are the primary methods of exploiting cache memory and extracting sensitive data (Yarom and Falkner, 2014; Gruss et al., 2016).

Prime + Probe: The attacker uses this mechanism to fill the relevant cache lines with data. After that, the attacker waits for the victim to execute various encryption procedures on the shared virtualized environment. Following then, the attacker measures the time for recovering previously loaded data. Thus, the attacker will be aware of the data that has been taken from the cache memory and will be able to identify the cache lines that were utilized in the victim's encryption operations as a result of this. Aside from the absence of shared libraries and page deduplication, this strategy is also quite efficient.

Flush + Reload: The attacker begins by flushing the appropriate memory lines from the cache. After that, the attacker waits for the victim to complete certain encryption procedures in the shared virtualized environment. Following that, the attacker reloads the flushed lines and records the time it takes them to get access. Using the timing information, the attacker can determine whether or not the victim has successfully retrieved the cache lines during the encryption process. This strategy makes use of shared libraries and memory deduplication to achieve its results.

Flush + Flush: The attacker first clears the relevant cache lines using flush instruction. After that, waits for the victim to complete specific encryption procedures on the shared virtualized system. Following that, the attacker flushes the prior cache lines again and monitors the time for the flush instructions to be executed, avoiding direct cache accesses. The attacker can employ this sort of attack to break a cryptographic key (Gruss et al., 2016). This strategy makes use of shared libraries and memory deduplication to achieve its results.

2.2 Cache Side-channel Attacks Implicit Characteristics

This section addresses the implicit characteristics of side-channel attacks that reveal the interior design of these attacks and how they work, as shown in Figure1. We focus on cache attacks. As (Irazoqui et al., 2016; Irazoqui et al., 2018) mentioned, there are several characteristics and instructions involved in the design of cache attacks.

All cache attacks have three main common characteristics, i.e., high-resolution timers, memory barriers, and cache evictions (Irazoqui et al., 2016; Irazoqui et al., 2018; Yarom and Falkner, 2014).

High-Resolution Timers: as shown in Figure 1 (line 7 and line 12), cache attacks depend on the difference time between retrieving data from the cache and retrieving data from the RAM, and also between retrieving data from different cache's levels. There-

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3
4    asm __volatile__ (
5      "  mfence             \n"
6      "  lfence             \n"
7      "  rdtsc              \n"
8      "  lfence             \n"
9      "  movl %%eax, %%esi  \n"
10     "  movl (%1), %%eax   \n"
11     "  lfence             \n"
12     "  rdtsc              \n"
13     "  subl %%esi, %%eax  \n"
14     "  clflush 0(%1)      \n"
15     : "=a" (time)
16     : "c" (adrs)
17     :  "%esi", "%edx");
18   return time < threshold;
19 }
```

Figure 1: Attack Characteristics snippet from (Yarom and Falkner, 2014).

fore, it is necessary to utilize an instruction that reads the time accurately, such as Time Stamp Counter (*rdtsc*), which provides high enough accuracy to differentiate between retrieval times.

Memory Barriers: as shown in Figure 1 (line 5, 6, 8, and 11), the file contains two instructions, *mfence*, and *lfence* to serialize all store and load operations that occurred before the *mfence* and *lfence* instructions in the program instruction stream. These instructions can be embedded into the attack scripts (x86 and amd64 instruction reference, 2019).

Cache Evictions: as shown in Figure 1 (line 14), in the beginning, the attacker usually evicts the required cache line out of the cache using *clflush* instruction and waits some time to let the victim retrieves this cache line. Thus, the attacker recognizes that the victim has used the same cache line by measuring the cache line retrieval time. *Clflush* instruction can remove the required cache line out of each cache level (x86 and amd64 instruction reference, 2019).

The presence of all or part of these characteristics may pose a risk to the shared virtualized environment in which the cache, some applications, and cryptographic libraries are shared. Therefore, it is essential to analyze executable files to explore these implicit characteristics that indicate that these executable files are malicious and prepared for attack.

3 PROBLEM DEFINITION

Due to increase in security threats to shared virtualized environments, it has become necessary to design integrated solutions to keep the shared virtualized environment protected from any cache side-channel attacks that might occur to it. We hypothesize that the shared virtualized environment needs a combination of dynamic and static analysis to ensure its safety, prevent from attacks, hence take advantage of these different analysis approaches. Static analysis is often known to produce high false-positive results ,and since checking the files inside the system requires a high load, it could affect the performance of the system. Therefore, this paper aims to explore a mechanism that supports static analysis by performing a dynamic analysis to detect suspicious behaviour within the virtualized environment and identify the suspicious VM. The static analysis is then performed to ensure the identification of malicious programs that could potentially protect side-channel attacks on the shared virtualized environment.The proposed mechanism receives the VM's ID to extract a set of the CPU performance counters readings then these readings are analyzed using machine learning classifiers.

If suspicious behaviours are detected then the VM is identified and the VM's name is extracted to examine the VM disk image for potential malicious programs. After this, the executable files are extracted to be disassembled and analysed so as to ensure the presence of the implicit characteristics of side-channel attacks. They are then classified into four classes from the highest threat to the lowest threat using neural networks. After these operations, a comprehensive view of the threat level of the VM is obtained, and then the malicious VM is then suspended depending on the neural network's classification outcome. These various and sequential analyses are done inside the host machine which can monitor the live performance of the VM, access the VM's disk image, disassemble, and analyse it to detect malware related to side-channel attacks. It can also suspend the VM or even eliminate it, thus ensuring a safe and attack-free shared virtualized environment. Using different types of analysis that support each other could potentially make the results more accurate and reliable.

4 METHOD

The proposed detection mechanism is achieved over several successive operations and stages.

Stage 1: The proposed mechanism receives the VM's process ID. It then utilizes Linux Perf to obtain the performance readings from the performance counters.

Stage 2: We create and train a logistic regression model that processes the data extracted from the performance counters and acts as a classifier between suspicious and normal behaviours.

Stage 3: If there is suspicious behaviour, we ac-

Table 1: Score-based threat classification.

Characteristics	Red	Orange		Yellow		Green		
	Case1	Case1	Case2	Case1	Case2	Case1	Case2	Case3
Clflush	✓	✓	✓	✓	✗	✗	✗	✗
Rdtsc	✓	✓	✓	✗	✓	✗	✗	✗
Mfence	✓	✓	✗	✓	✓	✓	✓	✗
Lfence	✓	✗	✓	✓	✓	✓	✗	✗

cess the VM's disk image on the KVM host by using the Libguestfs tools (libguestfs, 2019) that can also be used in popular hypervisors such as VMware and Hyper-V.

Stage 4: We then extract the executable files from the VM's disk and store them into a file to be checked in the stage number 6.

Stage 5: In this stage, we capture the VM's RAM status periodically using AVML Framework (Microsoft/Avml, 2020) and then analyze them from within the host using the Volatility Framework (VolatilityFoundation, 2020) to extract the files that have been stored in the RAM, thus gaining information of which files have been accessed recently. We then filter these files to reduce the number of files extracted from the disk and obtain more information of the files, such as files' paths to be processed later. This step is optional if only the RAM image of the VM need to be analysed.

Stage 6: We disassemble using the Objdump command in Linux and examine the extracted files against the implicit characteristics of the cache attacks codes. The implicit characteristics of the cache attacks codes are clflush, rdtsc, mfence, and lfence, as mentioned in (Irazoqui et al., 2018; Irazoqui et al., 2016; Yarom and Falkner, 2014). The result of this stage will be a dataset that will feed into the next stage.

The result is organized in columns as follows: File path, $clflush$, $rdtsc$, $mfence$, $lfence$, and $ThreatLevel$. The value of 1 indicates the existence of the implicit characteristic, otherwise it is set to 0 for each column. The threat level can take a value from 0 – 3 based on the combination of characteristics observed. For instance, we know memory barrier instructions ($mfence$, and $lfence$) are not a threat if they are not issued together with *rdtsc* timers or $clflush$ eviction instructions. We have designed the following score-based threat classification as shown in Table 1.

- Red: This level is considered the maximum threat and expresses the presence of all the implicit characteristics.
- Orange: This level is considered a high threat and expresses the presence of two implicit characteristics ($clflush$ and *rdtsc*) this may include having only one of the memory barrier instructions ($mfence$, and $lfence$).
- Yellow: This level is considered a low threat and expresses the presence of only one of the implicit characteristics of($clflush$ or *rdtsc*); this also may include having only one of the memory barrier instructions ($mfence$, and $lfence$).
- Green: This level represents the non-existence of a threat and indicates the absence of any dangerous instructions. At the same time, it may include memory barrier instructions ($mfence$, and $lfence$) that do not represent a threat.

Table 2: Intel Architectures Performance Monitoring Events (Intel, 2017).

Performance Counters
mem_load_uops_retired.l1_miss
mem_load_uops_retired.l2_miss
mem_load_uops_retired.l3_miss
br_inst_exec.all_branches
instructions
L1-dcache-load-misses
L1-icache-load-misses
LLC-load-misses
LLC-loads
cache-missess
cache-references
iTLB-load-misses
iTLB-loads

Stage 7: We create and train a neural network model using the SoftMax algorithm to classify executable files based on the previously mentioned threat levels.

Algorithm 1: Pseudo-code of the Dynamic Analysis.

Input: VM Process ID
Output: Malicious=1 or Benign=0
Receive *vm process id*
Model = *logistic regression*()
for *Infinite Loop* **do**
  Pause *for* 15 *seconds*
  Run *perf kvm stat −o output.txt −e event counters*
  Output = *open and read output.txt*
  while $fgets(line, size, Output) != Null$ **do**
    Convert *counters readings to integers*
    Save *counters readings to file.csv*
  end while
  Data = *open and read file.csv*
  Result = *Model* ← *Data*
  if *Result* > 0 **then**
    Detect *malicious behavior*
    Stop *ksm*
    Start *static analysis*
  end if
end for

5 EXPERIMENTAL RESULTS

We executed the experiments using the QEMU-KVM hypervisor on various hosts; Ubuntu 18.04.5 LTS had Intel Core i5-4200M CPU, Debian 10 had Intel Core i5-4200U CPU, and CentOs 8 had Intel Core i5-5300U CPU. We then created two VMs running an Ubuntu 18.04.5 LTS OS, one VM running as a victim VM, and the other running as an attacker VM. We installed the Libguestfs Tool (libguestfs, 2019), the Linux Objdump Disassembler, and Volatility Tools (VolatilityFoundation, 2020) inside the host. For the guests, we installed AVML (acquire volatile memory for Linux) (Microsoft/Avml, 2020) to capture the RAM status regularly. We designed the shared virtualized environment utilizing the QEMU-KVM hypervisor's default settings. The experiment consisted of two main parts as follows.

5.1 Monitoring Suspicious Behaviours

For monitoring suspicious behaviours, We used the Mastik framework to carry out the various cache attacks, such as Flush+Reload, Flush+Flush, and Prime+Probe attacks. Also, we used the Linux Perf tool to extract CPU performance counters readings from 13 counters every 15 seconds as shown in Table 2. We executed the experiments with four scenarios. The first scenario is normal activities where we considered the observations without any attacks or background applications. Secondly, the observations were taken during the normal noise of background applications with no attacks performed in this scenario. In the third scenario, we fetched the readings from the performance counters during the execution of the cache attacks without any interference from background applications. Finally, we carried out the attacks with several applications, and the readings were observed in this scenario as well.

We gathered the training data from various scenarios because other background applications running on the VMs affect the performance counters. We acquired data during cache attacks, both with and without running multiple background programs. About 3610 samples were collected. Also, a binary classification model was created using logistic regression to analyse the input data and categorize whether the system was under attack or not; 0 indicated a no attack state and 1 indicated an attack state.

We plotted the readings for the various aforementioned different scenarios. Figure 2 show the clear variance in instructions counters readings recorded in all scenarios. Also, all performance counters were analysed using machine learning to interpret the data more accurately and efficiently. We implemented the logistic regression classification model to analyse and classify the data based on whether the extracted data indicated the presence of an attack or not, in-line with the first step of the proposed detection process. The model was trained on 70% of the samples collected and tested on 30% of the samples. The model showed about 99% accuracy and F1-score 0.99 for the test case.

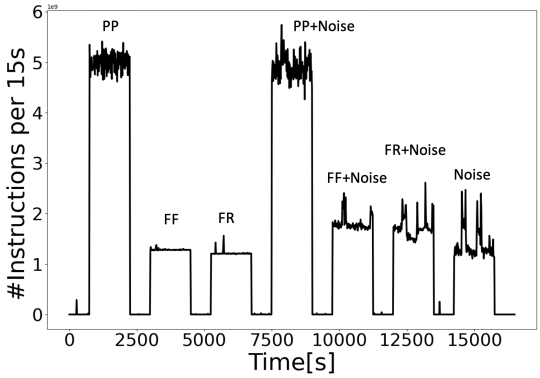

Figure 2: Experiment scenarios of all cache side-channel attacks in two cases without noise attack and with background noise. PP represents prime+probe, FF represents Flush+Flush, and FR represents Flush+Reload.

5.2 Static Analysis for VMs

Our static analysis involved the Mastik tool designed by Yarom et al.(Yarom, 2020), Xlate (Chiappetta et al., 2020), Cache Template Attack source code represented by Gruss et al.(Gruss et al., 2019b), and the Flush + Flush attack tool (Gruss et al., 2019a), as well as other Github repositories inspired by "Cache Template Attacks"(Gruss et al., 2015) and "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack" (Yarom and Falkner, 2014) such as (Nepoche, 2017), (Akash, 2018), (Pasic, 2019), (Park, 2018), and (Nagnagnet, 2018). We implemented the VM static analysis experiments in two scenarios as described below.

VM Disk Image Analysis: In this scenario, our proposed mechanism received the name of the VM. It created a mount point for the VM using libguestfs tools (libguestfs, 2019) After that, we extracted the executable files in both user and kernel space through command-line tools implemented using the *C* language and stored the resulting files' paths in a *.txt* file. We then used the Linux Objdump tool to disassemble the executable files and display the opcodes of the executable files in the assembly language. Next, we made sure of the presence of the attack opcodes that had implicit cache side-channel attacks characteristics inside the executable files. Finally, we recorded the

results of each extracted executable file and stored the results to be processed in the neural network model depending on the SoftMax classification to determine the threat level of these files.

VM RAM Image Analysis: In this scenario, we analysed the VM's RAM image. We performed this experiment in several steps. First, we downloaded and installed the AVML (Acquire Volatile Memory for Linux)(Microsoft/Avml, 2020) to periodically capture the VM's RAM image. In addition, we have downloaded and installed the Volatility Framework (VolatilityFoundation, 2020) to support analysing RAM images. Using these tools, our method was able to capture images of the RAM periodically, making it possible to track the operations of the VM. It was constantly updated during the operation of the VM. From the host device we were able to process and analyse the RAM and recognize the files stored in the RAM making our method more effective in detecting an attack.

We accessed the VM with suspicious behaviour inside the shared virtualized environment using the libguestfs tools (libguestfs, 2019). We then extracted the executable files in a similar fashion as the first scenario, after which we accessed the RAM image produced from the AVML (Microsoft/Avml, 2020) periodically to capture and extract the executable files. We then filtered the files by comparing the files' paths elicited from the disk image and the files' paths elicited from the RAM image to acquire sufficient information about the files' paths. We then analysed the filtered executable files using the Linux Objdump tool to convert the executable files into an assembly language, making it straightforward to investigate the presence of implicit attributes that comprises a threat on the system. We then stored the results to be classified later using the neural network model.

About 4,500 data samples were collected to train and test the model in two scenarios, namely the case of no attack files and the presence of a set of attack files. We used TensorFlow, a Google-provided open-source framework for machine learning methods, to develop the SoftMax classification model. We configured the training settings with 200 epochs and 10 as the batch size. In addition, we applied the Adam optimization algorithm to adjust the weights and minimize the loss. Furthermore, we included early stopping in our training through a callback. The model used as a classifier to classify the executable files according to the threat levels and The accuracy of the model ranged from 96% to 99%. We plotted the validation and training loss as shown in Figure 3.

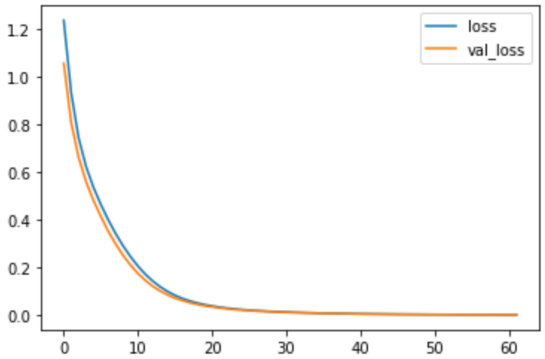

Figure 3: Validation and Training loss

6 EVALUATION

We evaluated the proposed method in terms of overheads and accuracy of detecting suspicious behaviours resulting from launching side channels and measured in different cases with a noise background from running several applications during the attacks. In the dynamic analysis, we produced the readings from the system every 15 minutes to classify them based on whether they indicated the presence of a side-channel attack. We then measured the classification accuracy while using the Linux Top tool to check the CPU usage and the system load during operations.

We tested 200 samples for each host involving various side-channel attacks. The system counters readings were extracted by the Linux Perf to be classified. We then classified these samples using the logistic regression classification. There were simple differences in the accuracy of detecting suspicious behaviour ranging from 96% to 99%. The CPU usage was between 0.11, 0.44 and the load on the system ranged from 0.6 to 3 for all host devices in the dynamic analysis, as shown in the Table 3.

We then conducted several experiments to measure the accuracy of the static analysis and measured the CPU usage and the load on the system. Different host operating systems were used. In each of them, we created a set of VMs by downloading and installing a set of side-channel attacks, tools, and executable files as described in Section 5. Attack files included a collection of attacks framework tools such as the Mastik framework, Xlate framework, and other attack project source codes. After that, we performed the static detection in two scenarios, the desk analysis, and the memory analysis. As shown in the Table 3, the proposed method detected attacks on executable files with 97-98% accuracy, with between 10–25% CPU usage, and between 0.85 –1.46 system overhead.

We designed a mechanism that depends on ana-

Table 3: Experiment results

Dynamic Analysis				
Os and CPU Type	%CPU	Loads	Accuracy	Time
Ubuntu i5-5200U CPU	0.6 - 3	0.33 - 0.35	%96.00	15 Sec
Debian i5-4200U CPU	1.3 - 2	0.11 - 0.22	%99.00	15 Sec
CentOS i5-5300U CPU	0.6 - 2	0.25 - 0.44	%95.58	15 Sec
Static Analysis				
Os and CPU Type	%CPU	Loads	Accuracy	Time
Ubuntu i5-5200U CPU	18 - 25	0.85 - 1.25	%97.91	8 - 12 Min
Debian i5-4200U CPU	10 - 24	1.09 - 1.46	%98.31	8 - 12 Min
CentOS i5-5300U CPU	12 -24	1.02 - 1.38	%97.27	8 - 12 Min

lyzing the behaviour of VMs at the beginning of the approach and then integrating it with static analysis to ensure good system performance, check the results, the process of verifying the results of the dynamic analysis is very important, especially in the case of false positives and then decide whether to suspend the VM or not. Based on the obtained results, we were able to identify suspicious VM behaviors, and scanned the VM against executable attack files with high accuracy. Also, this approach is applicable in the shared virtualized environments to monitor the activities of the VMs and identify the threat level of the suspicious VM from within the host machine.

7 RELATED WORK

Several previous approaches have been proposed for detecting the cache side-channel attacks. This section discusses the different approaches and identifies their limitations, inspiring us to address their drawbacks in our proposed method.

(Irazoqui et al., 2018) presented MASCAT, a mechanism used for microarchitectural attacks detection by static analysis of attacks' executable files. MASCAT utilizes static analysis to scan the attacks' elf files searching for implicit attacks' opcodes that are usually present in their design. However, MASCAT has a set of limitations that may hinder its adoption as a suitable solution for virtualized environments as it contains a high percentage of false positives. It also creates significant overhead in the system. Moreover, it is not usually used to detect and protect against malicious programs in the shared virtualized environment to scan the VM's disk and RAM. Furthermore, (Mushtaq et al., 2018) introduced a run-time detection mechanism for detecting cache attacks by monitoring hardware performance counters using Intel CMT (Intel Cache Monitoring Technology) to obtain performance readings, then analyzing the readings using a set of machine learning algorithms to classify malicious behavior. Also (Chiappetta et al., 2016) presented Flush+Reload attacks detection methods. Although they produced accurate findings, they were insufficient for identifying other types of cache attacks, e.g., the Flush+Flush attack.

(Bazm et al., 2018) presented a detection method based on Intel Cache Monitoring Technology (Intel CMT) and Hardware Performance Counters (HPCs). Moreover, it used the Gaussian anomaly detection algorithm to identify the attack status. Although this detection mechanism produced accurate results, it was adversely affected if there was noise(Mushtaq et al., 2018). Another study by (Cho et al., 2020) presented a machine learning method to monitor and detect the malicious behaviours of VMs that indicate cache attacks. The approach collected data for the model by utilizing Intel Performance Counter Monitor (Intel PCM) and then classified malicious behaviours and identified the type of the cache attack.

In previous our work (Albalawi et al., 2021) we have proposed a method for detecting cache side-channel attacks using memory deduplication. The method was used inside the VM and then classified suspicious behaviours using the shared executable file or shared cryptographic library. Our proposed method in this paper can also work in conjunction with the previous method to provide more robust and more reliable protection. The previous paper method detects the cache attack from within VMs when using sensitive executable programs such as shared cryptographic libraries.

Although various approaches for malware detection have been introduced, these approaches have important limitations. First, they perform static analyses frequently and do not require the start-up condition, thus increasing the load on the system. Moreover, they are also ineffective in detecting and protecting the shared virtualized environments against various side-channel attacks. Additionally, most of the previous methods need improvement in performance and results. Finally, none of the well-known antivirus tools detect any of the attacks we have analysed (cache side-channel attacks) (Irazoqui et al., 2018; Irazoqui et al., 2016). We address the limitations of the current detection methods by proposing an approach that monitors and detects any abnormal behavior of VM periodically and then performs static analysis of the detected VM's executable files and then classifying the threat level of VM's executable files using neural network classification algorithms, thus eliminating the malicious VM, and protecting the shared virtual environment with acceptable performance.

8 CONCLUSION

We have proposed a mechanism to detect and protect against Cache side-channel attacks. The proposed mechanism combines dynamic analysis and static analysis to detect the suspicious behaviour of the VM and then analyzes the executable files stored in the disk and RAM of the suspicious VM to recognize the implicit characteristics of the attacks. Our proposed mechanism combines the advantages of dynamic analysis and static analysis to reduce the load on a system. The proposed mechanism can detect attacks in the range of 96–99% accuracy and with 0.6–25% CPU overheads. In future work, we aim to improve and expand the analysis to include the other microarchitectural attacks to which the shared virtualized environments are exposed. We also plan to integrate the proposed mechanism with one of the well-known antiviruses to maintain a shared virtualized environment guarded against viruses and microarchitectural attacks.

REFERENCES

Akash, K. (2018). Flush-reload-attack. https://github.com/AkashWorld/Flush-Reload-Attack.

Albalawi, A., Vassilakis, V., and Calinescu, R. (2021). Memory deduplication as a protective factor in virtualized systems. In *Int. Conf. on Applied Cryptography and Network Security*, pages 301–317. Springer.

Anwar, S., Inayat, Z., Zolkipli, M. F., Zain, J. M., Gani, A., Anuar, N. B., Khan, M. K., and Chang, V. (2017). Cross-VM cache-based side channel attacks and proposed prevention mechanisms: A survey. *J. of Network and Computer Applications*, 93:259–279.

Bazm, M.-M., Sautereau, T., Lacoste, M., Sudholt, M., and Menaud, J.-M. (2018). Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In *3rd Int. Conf. on Fog and Mobile Edge Computing (FMEC)*, pages 7–12. IEEE.

Chiappetta, M., Savas, E., and Yilmaz, C. (2016). Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174.

Chiappetta, M., Savas, E., and Yilmaz, C. (2020). Xlate:. https://www.vusec.net/projects/xlate/.

Cho, J., Kim, T., Kim, S., Im, M., Kim, T., and Shin, Y. (2020). Real-time detection for cache side channel attack using performance counter monitor. *Applied Sciences*, 10(3):984.

Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2016). Flush+ flush: a fast and stealthy cache attack. In *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer.

Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2019a). Flush + Flush. https://github.com/IAIK/flush_flush.

Gruss, D., Spreitzer, R., and Mangard, S. (2015). Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912.

Gruss, D., Spreitzer, R., and Mangard, S. (2019b). Cache Template Attacks. https://github.com/IAIK/cache_template_attacks.

Intel (2017). Intel® 64 and ia32 architectures performance monitoring events. https://usermanual.wiki/Document/335279performancemonitoringeventsguide.2005880979/view.

Irazoqui, G., Eisenbarth, T., and Sunar, B. (2016). Mascat: Stopping microarchitectural attacks before execution. *IACR Cryptol. ePrint Arch.*, 2016:1196.

Irazoqui, G., Eisenbarth, T., and Sunar, B. (2018). Mascat: preventing microarchitectural attacks before distribution. In *8th ACM Conference on Data and Application Security and Privacy*, pages 377–388.

Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. (2014). Wait a minute! a fast, cross-vm attack on aes. In *Int. Workshop on Recent Advances in Intrusion Detection*, pages 299–319.

libguestfs (2019). Libguestfs tools for accessing and modifying vm disk images. https://libguestfs.org/.

Microsoft/Avml (2020). Microsoft/avml: Avml - acquire volatile memory for linux. https://github.com/microsoft/avml.

Mushtaq, M., Akram, A., Bhatti, M. K., Rais, R. N. B., Lapotre, V., and Gogniat, G. (2018). Run-time detection of prime+ probe side-channel attack on aes encryption algorithm. In *Global Information Infrastructure and Networking Symp. (GIIS)*, pages 1–5.

Nagnagnet (2018). Prime+Probe is a last-level cache side-channel attack. https://github.com/nagnagnet/PrimeProbe.

Nepoche (2017). Flush and reload cache side channel attack. https://github.com/nepoche/Flush-Reload.

Park, J. (2018). CSCA (Crypto Side Channel Attack). https://github.com/jinb-park/crypto-side-channel-attack.

Pasic, H. (2019). Side channel attack (cache attack. https://github.com/HarisPasic/SideChannelAttack.

Saxena, S., Sanyal, G., Srivastava, S., and Amin, R. (2017). Preventing from cross-vm side-channel attack using new replacement method. *Wireless Personal Communications*, 97(3):4827–4854.

VolatilityFoundation (2020). Volatility framework - volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility.

x86 and amd64 instruction reference (2019). Core instructions. https://www.felixcloutier.com/x86/index.html.

Yarom, Y. (2020). A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik/.

Yarom, Y. and Falkner, K. (2014). Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732.