TrapMP: Malicious Process Detection By Utilising Program Phase Detection

Zirak Allaf
School of Computing
University of Portsmouth
Portsmouth, UK
Email: zirak.allaf@port.ac.uk

Mo Adda
School of Computing
University of Portsmouth
Portsmouth, UK
Email: mo.adda@port.ac.uk

Alexander Gegov
School of Computing
University of Portsmouth
Portsmouth, UK
Email:alexander.gegov@port.ac.uk

Abstract—**Hardware and software have failed to securely manage the sensitive elements of cryptographic algorithms in computational environment due to memory contentions. This opened new opportunities for hackers to carry out side channel attacks on a system and steal sensitive data. Existing Side-channel attack techniques show that attackers can exploit the microarchitecture and OS vulnerabilities. The recent Meltdown [1] attack for instance, using Flush+Reload technique, exploits program execution attributes such as "out-of-order execution" to break the logical isolation between the memories and processes. In this paper, we have developed a real-time detection and identification system against side-channel attacks. Unlike previous works, the proposed approach does not rely on synchronisation between the attackers and victims. This is realised by taking a course of program phase analysis, through performance counters, to extract Malicious Loop (ML). Simulation has shown that the proposed approach attained higher accuracy for up to 99% and efficient detection of Flush+Reload activities, through classification methods. Furthermore, the detection process, in native and cloud systems, unlike others, takes shorter execution time without additional costs, and the model benefits from very low overhead performance of approximately less than 1% of the host system.**

1. Introduction

Nowadays Internet and Cloud computing are increasingly in use on various devices ranging from smart devices to computer servers. Data protection becomes a great challenging factor. Although crypto-algorithms have emerged as promising approaches to protect data, hardware and software vulnerabilities attract hackers to target cryptographic components and steal their secret elements by achieving the side channel attack. The side channel attack is the action of stealing sensitive information by exploiting hardware/software vulnerabilities to provide unauthorised communication channels across independent entities in shared systems. The main attack characteristics can be identified firstly by relying on hardware contentions - cache misses to observe victim' activities, secondly by performing operations on the system without privileges, and finally by monitoring the processors' cycles [2] and the power consumption [3] as metrics to measure latency.

Various techniques have been employed in the literature to achieve side channel attacks, such as Flush+Reload [4], Prime+Prob [5], Flush+Flush [6] and Evict+Reload [7]. In this paper, we focus on Flush+Reload attacks. These schemes allow attackers to exploit the vulnerabilities at micro-architecture level by installing hidden communication channels through memory levels between the victim and attacker programs. One of the most common vulnerabilities is the hardware contention, which occurs in real-time during data access across multiple independent programs. Hardware contentions are series of threats operating on the sensitive data and thus reducing the degree of memory isolation in real-time across concurrent programs. The exploitation of hardware contentions has been practised at all memory layers including L1I [8], L1D [9], L2 [10], LLC [4] and main memory [11]. The attackers have also practised the aforementioned techniques against various cryptographic algorithms AES [2], DES [12], RSA [4], [13]. Recent studies showed the successful attacks occurred in security settings - hardware settings [14] and disabling OS features like page sharing [15] and KASLR [16], securing software implementations constant programming [17], hardware implementation for sensitive data SGX [18], and compiler optimisation [19].

However, past research shows that many detection and mitigation solutions have been proposed to eliminate such attacks, such as detection system [20] [21] [22] and protection [23] [24] [25]. However, the attackers continuously explore new vulnerabilities [1] [26]. The exploitation has reached a level where the data protection in kernel space across user programs has been compromised to side channel attacks. The most recent attack is Meltdown [1] in which the attacker uses the Flush+Reload technique to exploit the execution mechanism "Out-of-order execution" to reveal logical memory isolation of the sensitive data in kernel space.

The above problems indicate that it is crucial to develop a detection technique to efficiently mitigate side-channel attacks from stealing sensitive data. In this paper, a new knowledge-based framework is proposed which can leverage hardware features to analyse process activities at a low level (processor core) with the aim of detecting malicious processes which may lead to achieving side-channel attacks in the user space. The framework is capable of eliminating security threats against cache memories, which accommodate the cryptography algorithms. The proposed framework uses a new profiling technique which observes processor core level activities. This profiling allows the use of machine learning algorithms to build a classification model, which does not require any synchronisation between the victim and attacker programs to detect side channel attacks in the system. To the best of our knowledge, most of the detection systems rely on synchronisation to analyse data dependency in to detect the attacker. Nonetheless, there are a number of factors having a negative impact on the synchronisation approach, such as heavy workloads leverage [27] and program execution instability [21]. Particularly in cloud systems, the synchronisation approach requires more investigation to recognise the attack pattern [28]. Whereas, in the proposed framework, detecting the attack in both native and cloud systems have the same cost.

The rest of this paper is organised as follows. Section 2 presents a brief review of the related work. In section 3, the general layout of the framework is presented. Section 4 elaborates the proposed detection and identification methods of the proposed framework (TrapMP). Section 5 demonstrates the simulation results with detailed analysis. Finally, a summery of the work is given in section 6.

2. Related Works

Side channel attack detection have been researched for about two decades. Various techniques have been used to detect, protect and prevention in various levels. Thus, this work discusses the review of recent work to reveal the most up-to-date best practice in side channel attack detection, and addresses limitations in previous work on the subject.

Profiling: Zhang et al. [29] and Payer [20] proposed the use of the `perf` tool against side channel attacks by monitoring existing processes in the system. [29] selected VMs at random to monitor, while [20] monitored all processes in the system. The fact remains that side channel attackers can escape observation because the use of `perf` in both studies depends on the file system `proc` to retrieve information about the system's existing processes. A case study on Malware attacks [30] demonstrated the feasibility of an attacker modifying the `proc` file to hide its process id from the system so then `perf` gained no information about the malicious processes. In this paper, on the other hand, we have proposed a system profiling mechanism that targets the processor cores instead of the `proc` file system, so then all program execution transactions appear on the observation. This means that the attackers cannot escape observation.

Native and Cloud systems: Alam et al. [31] and Chiappetta et al. [32] proposed detection systems using the machine learning approach to detect side channel attack in native systems. However, they failed to detect malicious VMs in cloud systems. Our work proposes a detection mechanism of ML in both native and cloud systems without an additional cost concerning the cloud systems.

Synchronisation: When the side channel attack uses hardware resources such as CPU cache memory, it basically relies on memory contentions in the repetition manner which leads to unintentional contentions. The attackers are unaware of this, and this causes significant abnormal activities. This can be easily detected by utilising a synchronisation approach. This means that the attack processes can be detected by relying on the data collected by the victim [22]. This approach is vulnerable in two potential circumstances. First, Allaf el at. [27] studied a comparison of multiple machine learning algorithms, namely age is SPEC cpu2006 int and fp application, in order to stress the CPU cache memory. As a result of this, heavy workloads have a negative impact on the detection accuracy of three machine learning algorithms including DT, PCAANN and KNN. All algorithms performed well when no workload was running, with int applications such as gcc and bzips degrading the accuracy. With fp, the accuracy got worse.

Performance: The performance overhead is the central issue affecting system performance in reference to any potential detection methods. System overheads need to be considered, and can be classified as either an OS-based overhead or an application-based overhead. An OS overhead is much more expensive than an application overhead. [33] recommends injecting a machine learning algorithm into OS scheduling to monitor CPU component usage to detect malicious processes. Cloudradar [29] employs and dedicates three processor cores to monitor malicious processes. Payer [20] continuously monitors all existing processes. These mechanisms incur overheads in the host system. In the proposed framework, instead of injecting machine learning algorithms at the OS level, the detection system is placed in the user space and only data collection is placed in the kernel space. The proposed framework does not monitor the whole processes in the system, and it instead profiles the processor core executions. Finally, the proposed framework does not require dedicated hardware and OS configurations.

3. Background

3.1. Real-time Scheduling

Scheduling is one of the core OS services to support and mange hardware resources across running programs. The main goal of scheduler is to minimise power consumption [34], which is used by the resources, and offer the optimal performance by minimise stalls [35] to provide the optimal dynamic adoption Dynamic voltage and frequency scaling (DVFS)[1] [36]. Thus, OS designers and researchers intend to propose optimal scheduling algorithms to aid bottlenecks and reduce power consumption in order to utilise highest possible speed that a CPU has with considering hardware limitations. The main focus in scheduling studies is the

1. is the adjustment process of power and speed settings on various processors in computing device

usage of underlying hardware resources efficiently and how to virtualise and share them across processes. On the other hand, side channel attacks come into account to distort the beauty of scheduler by misusing the shared resources due to scheduler vulnerabilities while using them [1], [26]; and become an obstacle in front of them to scale up CPU components as CPU speed.

3.2. Program Phase Utilisation

Program phase has been utilised in various computational problems related to performance, such as saving energy [34] and performance tuning [35]. Recent studies have found that the use of program phase leverages CPU scaling, which relies on the correlation between memory and CPU workloads. Program phase provides information to improve scheduler algorithms in the OS. For instance, Skrenes et al. [37] employed the Dynamic Voltage and Frequency Scaling (DVFS) mechanism, which balances high speed CPU with memory access latency to avoid stalls when the workload intends to fetch data from the cache memory. Furthermore, Zhang et al. [34] used dynamic configuration CPU frequency to save CPU power. This guides the system to switch the CPU frequency mode into a higher frequency rate when intensive CPU usage is indicated and vice versa. In addition, the program phase has been utilised in performance simulation tools to reduce the simulation time for benchmarks.

3.3. Program Phase Definition

In the computational environment, the total computation of any program can be divided into a set of intervals. Each interval is a slice of the program's execution. A set of intervals is composed to form a phase. The phase can occur multiple times within the program's execution. The transactions between two consecutive phases is called phase change [35]. Program execution behaviours vary from program to program including large-scale ones [38]. Thus, Dhodapkar et al. [39] categorised program phases into stable phase and phase changes. Stable phases are indicated if two or more phases have exactly similar activities, otherwise phase changes are indicated. The presence of phase changes indicates that the phase has been incurred with computational noise. Furthermore, phase detection relies on the nature of a program. The program may be a memory transactions or instruction stream [35], [40]. In addition, in some circumstances, it is hard to distinguish phase transition; however, selecting relevant events has a positive impact on phase detection by signalling the transactions between phases [40].

3.4. Malicious Loop Phase Modelling

Recall that the main part of a FLUSH+RELOAD attack program body is a malicious loop (ML), which steals secret key in AES; inside the ML, the two consecutive tasks are executed, flush a targeted memory addresses, which are the range of the memory addresses in which the AES look-up table is stored, using `clflush` instruction and are followed by access to the flushed address continuously. Thus, in the ML structure, each phase has a set of intervals of similar execution activities. The activities are the consumption of the hardware resources such as the L1, L2 and LLC caches. Any `clflush` instruction causes an equal number of cache misses at each hierarchical cache level, when the next access is achieved. As L1 and L2 are private per core, high frequency context switches have more influences on L1 and L2 misses than LLC. This leads to the visualisation of ML phases by noting that LLC cache misses have clear phase transactions between two adjacent phases. Figure 1 depicts the complete program phase of Flush+Reload from start to end in user space by capturing E_1 and E_2; and in this case the Flush+Reload program runs for a short period of time for the presentation purpose. However, when utilising Flush+Reload in real systems, the loop boundary is much longer in order to retrieve the entire key bits (as described by [15]). Figure 1 - (a) represents E_1 cache misses in a more organised way than for Figure 1 - (b) which is E_2.

However, the phases of a program are decomposed into sub-phases in execution-time. Recall that each task of the program is fragmented into jobs, represented in sub-phases, and shuffled with jobs of other programs, so that the jobs are queued by scheduler for execution; and the task scheduler fairly distributes them across online processor cores. Each of such sub-phases appears in between other sub-phases of other workloads in user space. Figure 6 shows the sub-phases of the Flush+Reload program which are distributed across other sub-phases of other programs in the same processor core's execution timeline. The sub-phases of the ML can be recognised across sub-phases of other programs, but the points that indicate the phase transition are hard to capture. HPCs can virtualise sub-phase transactions between two different sub-phases of two independent programs by relying on the ML behaviour based on their execution attributes. This transaction can be used to extract the ML activities across existing workloads in the computational environment. However, defining sub-phases is not an easy task in heterogeneous workload due to changes in program behaviours during run time which are leveraged by dynamic hardware adoption and configuration, but selecting the most relevant and efficient event to characterise ML leads to distinct ML sub-phases across other workloads.

3.5. Threat Model and Assumptions

This section illustrates the potential Flush+Reload attack on microprocessor caches. The attacker exploits hardware and OS vulnerabilities by utilising intentional hardware contentions with a victim's processes, while both the attacker and victim are synchronised, in shared environments in which hardware resources, such as CPU caches, are fairly shared across running applications in both native and cloud systems. The attacker and victim use an AES algorithm to encrypt plain text. An AES algorithm is implemented in `crypto.so`, which is a shared library in an OpenSSL package and it is installed in the host OS Ubuntu 14.04.

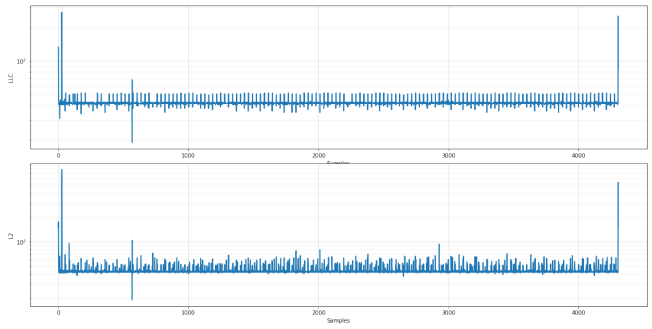

Figure 1. Signature of the attacker program in the native system shows the behaviour of the Flush+Reload program and how it interacts with underlying hardware during its execution

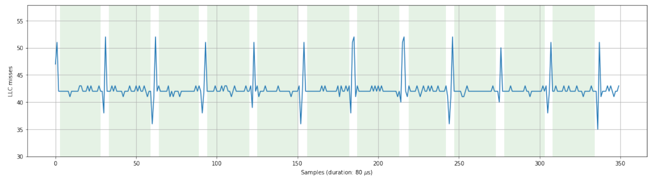

Figure 2. Signature of the attacker program in the native system shows the behaviour of the Flush+Reload program and how it interacts with underlying hardware during its execution and transparently provides interfaces to access HPCs. It is assumed that no malicious bodies have access to the PHCs to modify settings and distort the observations.

The attackers can be a malicious program in the host OS or VM in the guest OS. The attacker analyses the hardware cache contentions to deduce the AES secret keys.

4. TrapMP

TrapMP is a trapping method for capturing Malicious Processes (MP) at the processor core level. The TrapMP is composed of two parts: the detection and identification phases. Both phases rely on the usage of HPCs to support their models in detection and identification tasks. The detection model is responsible for detecting ML activities in the system, whereas the identification model is responsible for identifying the owner of the ML program. They request information from the kernel module about the state of processor cores, because both the detection and identification models run in user space; and programs in user space have no access to HPCs. Figure 3 illustrates the high level of the framework and shows its main components, along with their hardware usages and their communications. The yellow notations represent the whole process in chronological order.

Detection Phase: In this phase, the detection model is responsible for detecting side channel attacks, namely Flush+Reload in the system. The model utilises supervised machine learning algorithms to classify the attack activities which are achieved by the attacker program in user space. The detection model continuously observes program execution attributes on active processor cores from any ML activities. ① The Detection Agent (DA) sets up the communication channels with the Event Recorder Agent

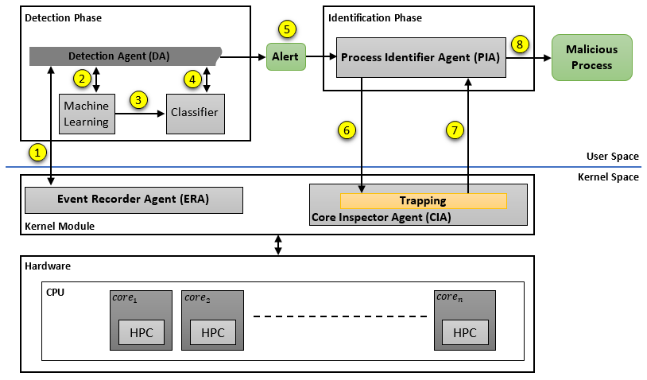

Figure 3. An overview of the proposed framework (TrapMP)

(ERA) in kernel space to request S samples per processor core. ② Then DA performs prepossessing of raw data by applying shift and aggregation mean function to combine n of consequent samples to capture ML sub-phases. ③ The DA feeds the new data-sets to the classifier to extract the attacks pattern. ④ The classifier sends back the results to the DA. ⑤ The DA sends an alert to the identification phase if attack activities have been detected.

Identification Phase: is responsible to identify the attackers by setting up a trap routine to redirect the malicious program execution path to an interrupt routine. In the cloud settings, the framework is capable of trapping the malicious VMs and identifying them as having the same cost as the native system. ⑥ The Process Identifier Agent (PIA) requests the Core Inspector Agent (CIA) to inspect any program execution attributes related to the ML execution attributes. This is done by settings and initialising HPC counters to investigate the state of each processor core. ⑦ If the values of the PMC counters match the attack patterns, which relies on the statistical analysis model, then the CIA will trigger an interrupt to suspend the MP and yield the necessary information about the MP. ⑧ The CIA reports back details about the identification of the ML to the PIA, Finally, the PIA reports back on the identity of the malicious processes or VMs to the admin users.

4.1. Experiment Setup

The experiments were achieved on a HP Proliant DL360 G7 with Intel Xeon X5650 2.66GHz processor and 16 GB of RAM. The operating system is Ubuntu 14.04. Furthermore, SPEC cpu2006 benchmark has been used for testing purposes.

4.2. Feature Selection

In this study, feature selection plays the key role in detecting side channel attack by profiling processor cores in Real Time Systems (RTS), in which program execution attributes are captured during their assignment to the processor cores, because no information is provided about processes which are assigned to the processor cores such as PID. Proper features, which represent the execution attributes, supports classification algorithms to extract Flush+Reload attack activities with high accuracy. Thus, it is crucial to select features which makes the distinction between attack program and other workload in the system. Unlike previous work [20], [21], [29] in which the feature selection relies on the facts that relevant to the synchronisation between victim and attacker programs, in which features are selected based on the data dependency which indicates the co-relation between victim and attacker programs. Whilst, feature selection in this paper does not rely on synchronisation and data dependencies, instead it focuses the unintentional memory contentions by the attacker program. Table 4.2 presents the relevant cache misses to Flush+Reload attack.

PMCs	Anno.	Events
Programmable	E_1	LLC Misses
	E_2	L2_RQSTS.ALL_CODE_RD
	E_3	L2_RQSTS.DEMAND_DATA_RD_HIT
	E_4	L2_RQSTS.ALL_DEMAND_DATA_RD

TABLE 1. RELEVANT EVENTS TO SIDE CHANNEL ATTACK

4.3. Moving Window Aggregation (MWA)

The aggregation mean function is employed to find the related samples which belong to a single job in ML. As a scheduler cannot be controlled in terms of assigning jobs to the online processor cores and the duration of the assignments, the default is to guess how many samples belong to a job and for how long a processor core holds the job. Consequently, giving raw data to the machine learning algorithms will negatively impact on the performance of the classifiers in detecting side channel attacks [41]. Thus, we leveraged the MWA algorithm. To construct the phases of the ML by finding the set of consequent samples in execution time, which belong to the ML jobs.

MWA is the process of partitioning the data-set D which has N samples with F features into subsets $\bar{D}$. Each subset contains the consequent n samples, when $n \subseteq N$, and F features; and averages them to produce one sample which represents one ML phase. As a result, a new data-set $\bar{D}_i$ will be generated with the length of $\frac{n}{N}$ samples. This is to transform each the ML phases n into one sample and this will be classified as attack activities. Still it is not guaranteed that the whole body of the phase will be captured, because there might be a sample from the neighbour jobs of other workloads which will interfere with the ML phases. To overcome this problem, the original data-set will be shifted n times and the same procedure split and the mean function for each subset will be repeated to generate n of $\bar{D} = \{\bar{D}_1, \bar{D}_2, .., \bar{D}_n\}$, where n is also the threshold which indicates less than the maximum length of the ML samples which might appear in each ML phase in real-time. The whole data-set will be given to the classifier to allow more chance to detect any potential ML activities. The MWA algorithm provides reliability and robustness in the detection system, because it tries to extract and capture each ML phase by combining the chronological sequence of samples which belong to each ML job in execution-time.

4.4. Methodology

Bagging (short for Bootstrap aggregating) was introduced by Breiman [42] is an ensemble learning technique to decrease the variance of a predictor by bootstrapping samples with replacements from the original data-set to train prediction models of any supervised machine learning algorithms and aggregating their results to select the best predictor. Random forest is the implementation of bagging technique [42] to construct a collection of Decision Trees. Breiman [43], for the first time, introduced Random forest to decrease variance, which is generated by a single tree-based predictor CART, by constructing many internal tree predictors in the forest each of which is trained on an independent random sample derived from the original data-set with replacements. Furthermore, each random sample is composed of random features to increase the chance of contributing the maximum number of features in the splitting processes, which is called diversity. Random forest has many applications for balanced and imbalanced data-sets. It has received a lot of attention from researchers in imbalanced data-sets because Random forest encourages diversity [44].

4.5. Model Evaluations

After building the Random forest classifiers, we need to make sure that the models are efficiently applied in the unseen data-set. The evaluation metrics for classification models rely on confusion matrix, which contains information about the predicted classes produced by the classifier models and the actual classes from the original data-sets. When **True Positive (TP)** is the case where the classifier correctly recognises the positive samples in the data-set. **False Positive (FP)** in this case, the classifier miss-classifies the positive classes as negative. **True Negative (TN)** represents the total number of the negative classes detected by the classifier correctly. **False Negative (FN)** when the classifier miss-classifies the Negative samples as Positive. Based on the confusion matrix, we can derive the following metrics, which are used to analyse the performance of the Random forest classifiers.

1) **Recall/Sensitivity (True positive Rate)** corresponds to the TP samples which are correctly classified as positive.

$$Recall = \frac{TP}{TP + FN} \quad (1)$$

2) **Specificity (False Positive Rate)** corresponds to the FN samples which are incorrectly classified as positive.

$$Specificity = \frac{FP}{FP + TN} \quad (2)$$

4.6. Experimental Design

The design of the experiments of this paper can be summarised in to tow phases. In the first phase, for each of the two imbalanced data-sets, which are collected in both native and cloud systems, 20 different runs of ten-fold cross validation (CV) were executed. In CV, each new data-set is constructed from different data points in the original data-sets for both training and testing data-sets so that all data points contribute in the learners' building stage. For each iteration of CV, %70 of the original data-sets were used for training data-sets and the rest were for testing data-sets. In the second phase, bagging algorithm random forest algorithm has been used. With each CV iteration, the new training data-set is fed to each algorithm to build a classifier and then the new testing data-set is used to evaluate the classifier.

4.7. Experimental Results and Analysis

In this section, the results of the experiments are shown for Random forest algorithms and they are visualised by utilising (ROC) Area Under curve (AUC) ROC-AUC. Figures 4 and 5 depict the classifiers performance in discriminating between two process activities which are normal and attack.

In ROC-AUC figures, the classifiers outputs is represented as ROC curves, which represent the sensitivity (recall) and specificity calculations at incremental thresholds between zero and one across 10 folds when the same data-set is randomly shuffled, resulting in each fold having a different spread of the data. The Y axis plots the classifier output's True Positives Rates (recall) and the X axis plots False Positive Rates (specificity). Each fold is an individual ROC and is the light blue line. It represents detection quality. The solid blue line is the calculated mean. The ideal representation is when the ROC curves has x=0 and y=1. This indicate that the classifiers 100% classify normal and attack classes in unseen samples.

Figures 4 and 5 show the ROC metric that evaluates the random-forest classifier's ability to detect the ML activities among normal workloads in the host system in both native and cloud settings respectively. Success in observing program execution attributes and classifying processes as normal or attack as a measure of the risk of existing side channel attack in the system is shown as estimated by the AUC of ROC. The model identifies ML in a native system with very high accuracy (AUC=0.99 for an average of 10 folds, with a zero confidence interval), as it is shown in Figure 4. In the cloud, however, the same algorithm, when trained on a data-set that captured VM activities, was less accurate at predicting malicious activities from among other workloads (AUC=0.99, confidence interval=0.01), as shown in Figure 5. The classifier therefore has the same efficiency at identifying malicious loop activities in native and cloud systems. The noise incurred by L1 and L2 cache memories, which arises from the additional translation layer imposed by Structure as a Service (SaaS). Random forest utilises a bootstrapping technique 4.4 in which all data points in the

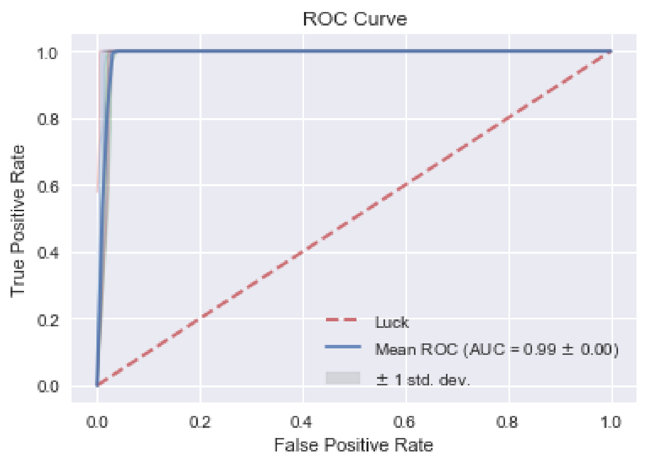

Figure 4. ROC-AUC for bagging algorithm (random forest) in native system

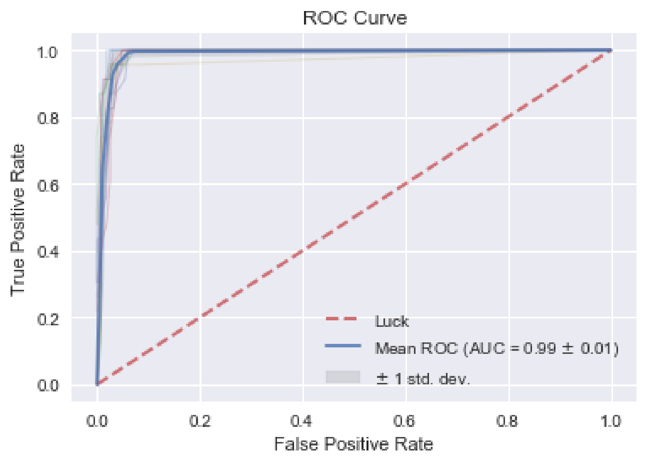

Figure 5. ROC-AUC for bagging algorithm (random forest) in cloud system

data-set are involved in model building, particularly in the imbalanced data-set. Randomforest tries to generate implicit balanced data-sets by bootstrapping the original data-sets; in each data-set, the minor class (attack class) is always placed first, followed by the major class for both training and testing sets, in which case the model is well-trained on the training and testing sets to eliminate under-fitting problems.

4.8. Performance

This section reports on the performance overhead which is generated by the detection model. In this experiment, the SPEC CPU2006 benchmark was running for about 13 hours with and without the detection model. The detection model was running in user space and continuously communicating with the Event Record Agent (ERA) to collect data and feed the detection model for malicious activities.The results suggest that the detection model has a very low impact on the performance of the host system; even in the worst case, the performance overhead is within 0.03.

5. Identification Phase

This section describes the process of identification malicious processes in host OS. This phase responsible in identifying the malicious processes which initiate the malicious loop inside side channel attack program.

5.1. Identification Model

Figure 3 depicts the whole process of detection and identification. From step ⑤, the Process Identifier Agent (PIA) is the entry point for identification. Let's assume that the identification model has received a message from the detection model. PIA acknowledges the Core Inspector Agent (CIA) to initialise the Trapping procedure. The CIA is a kernel module which is driver-based and where the trapping procedure listens to an incoming message from the PIA. The CIA starts to configure PMCs and initialises parameters including the thresholds. As it receives the message, the trapping procedure initialises a `for_each_online_cpu(cpu)` to examine the online processor cores. This function returns a cpu parameter used by the function `msr` to confirm that the function reads and it is then pinned to the specific processor core. This is done because, without using `wrmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h)`, there is no guarantee merely from using `rdmsr` and `wrmsr` instructions that the targeted processor core will be read. The CIA then examines each processor core individually to find the core serving the attack program.

A vector of PMC variable $vPMC[pc]$ is created, when $pc =$ the number of PMC, to store PMC counter values. In the inner loop of the identification algorithm 1, in the first iteration, the values of PMCs are stored into $vMPC_0$. In the other iterations, the new captured PMC is averaged with the $vPMC$ content. Until the counter reaches the length of the $phase$. The $phase$ variable indicates the minimum length of samples which might occur within one job. If, say, the number of samples is five then the loop inside the identification procedure takes five samples and checks the attack pattern by using the threshold parameters, which indicates the length of the MP phases; if there is no match between the $vPMC$ and the attack thresholds, the loop resets $vPMC$ and continues checking. If, on the other hand, the $vPMC$ values and the threshold match, this is the process that is causing the attack and the PMC counters are immediately reset to -1 to force a PMC overflow using the current process core. Recall, PMC interrupt is enabled, The PMC counter overflow causes the OS to suspend the current process assigned to the current processor core and hands control to the Trapping interrupt handler. Inside the Trapping interrupt handler, information about the suspended processor core is taken from the Processor Control Block (PCB) and passed back to PIA, which will now have the identity of the malicious process and can take necessary action to find its owner.

5.2. Identification model Evaluation

This section discusses and evaluates the results obtained from the experiments. To evaluate the identification, three experiments are conducted in which the only difference is the profiling settings. What is changed is the number of samples $\{5, 10, 20\}$ taken by the trapping task. The number of samples is critical to identification, because more than the threshold causes the identification model to miss the attacker; or less than the threshold for the identification model leads to the generation of False Negative due to inferring non-attack samples.

Figure 6 shows profiling with 1000 samples for each native and cloud Flush+Reload program. The duration of Flush+Reload program jobs running in native and cloud are different. The time quantum for the cloud-based jobs is longer than for the native ones. This duration has an impact on identifying the malicious process activities, which are denoted in the red and blue horizontal lines. Green boundaries show correct detection of the attack program by the algorithm; red boundaries show a failure to capture the attack program. The boundaries are not equal due to the soft scheduler, in which there is flexibility for the jobs to be completed. By relying on the analysis in Section 3.2, we can define the minimum and maximum required time to complete a job. Figure 6 shows the difference between the quanta for native and cloud jobs. Sub-figure (a) shows the scheduler for real-time executions in user space for a native system. Sub-figure (b) shows the scheduler for real-time execution of VM and host real time programs in user space. The VM job has a larger quantum than the native-based jobs. This is because recent work shows that the time quantum for jobs for VM processes is longer than for jobs in a native system for performance purposes. Thus, the identification model has more confidence in detecting malicious VM than a native-based malicious program.

Algorithm 1 Identification Algorithm

```
1: procedure IDENTIFICATION()
2:     $threshold, phase, counter$
3:     for each core: $c$ in $C$ do
4:         $obs = read(PMC)$
5:         if ($obs$ satisfied $thresholds$) then
6:             $counter++$
7:             if $counter > phase$ then
8:                 /* $ML$ is identified and force
9:                 its $MP$ to interrupt handler */
10:                $modify\_PMC(PMC = -1)$
11:            end if
12:        end if
13:    end for
14: end procedure
```

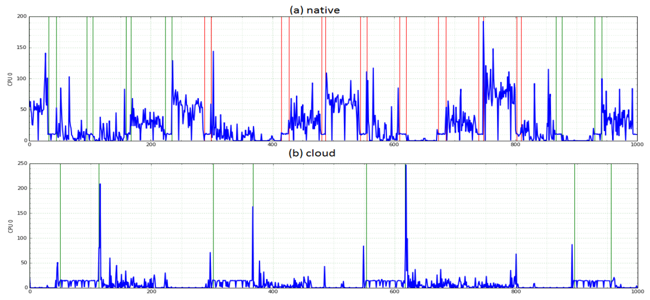

Figure 6. The execution time line is sliced among the attacker in a native system and 4 SPEC workloads. The malicious loop inside Flush+Reload program phases for LLC cache misses appear as chunks of samples which can be observed as process transactions on a specific processor core.

5.3. Discussion

In this section, we have reported on experiment results reflecting on issues which have an impact on the performance of detection and its robustness in detecting side channel attacks in both native and cloud systems. Regarding robustness, the profiling mechanism is not able to recognise the phases of the ML efficiently; instead, it is an auto mechanism to capture the program execution attributes. Thus, the MWA algorithm is used to extract the phases of the ML by aggregating the samples of a phase and then moving the entire data-set to inspect any possibilities of attack activities in the system. So, the profiling ensures that all the program execution activities are captured because it is auto capturing does not rely on any means to get information about each process . Instead, the identification mechanism is used to acquire the identity of the attacker. Another benefit of this approach is that the monitoring of program execution activities for native and cloud processes use the same process. This is because native and VM processes are executed concurrently using the same hardware resources (e.g. CPU). In this case, the same analysis is used for activities in both native and cloud systems with a slight degradation in the cloud system due to an extra translation layer in hypervisor in cloud systems. Furthermore, the detection models can identify more than one potential Flush+Reload attack in the system without having any effect on detection accuracy because multiple attacks are independently acting in the system and they never overlap or interfere each other. Thus, they are monitored independently.

Besides this, the identification phase relies on an interrupt and the identification model is executed only if the detection model detects an attack in the system. Consequently, any mis-classification will cause interrupts. The more interrupts, the more significant the performance overhead which is generated in the system. Thus, it is essential that the classification model be sensitive to correctly detect potential attacks.

6. Conclusion

This paper has proposed detection of side channel attacks using bagging technique. The paper also put forward a new profiling technique to captures the program execution attributes at core level. Thus the attacker cannot escape from the profiling in both native and cloud system. The ROC curve is used to evaluate the efficiency of the proposed classifier for the detection of side channel attacks. The classifier detects side channel attacks in both native and cloud systems with performance of up to 99% under SPEC CPU2006 workloads. However, the proposed method cannot detect techniques such as Prime+Probe due to the behaviour of the malicious loop inside the program. The future work will be devoted to the design of a model that can detect other side channel attacks such as Prime+Probe, Flush+Flush and Raw-hammer.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[2] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[4] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA 2006*. Springer, 2006, pp. 1–20.

[6] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: A fast and stealthy cache attack," *arXiv preprint arXiv:1511.04594*, 2015.

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[8] O. Aciiçmez, "Yet another microarchitectural attack:: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*. ACM, 2007, pp. 11–18.

[9] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[10] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 29–40.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[12] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards." *Smartcard*, vol. 99, pp. 151–161, 1999.

[13] S. Weiser, R. Spreitzer, and L. Bodner, "Single trace attack against rsa key generation in intel sgx ssl," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018, pp. 575–586.

[14] T. Kim, M. Peinado, and G. Mainar-Ruiz, “Stealthmem: system-level protection against cache-based side channel attacks in the cloud,” in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 189–204.

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.

[16] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, “Kaslr is dead: long live kaslr,” in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 161–176.

[17] T. Pornin, “Bearssl: A smaller ssl/tls library,” 2016. [Online]. Available: https://bearssl.org/

[18] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: Sgx cache attacks are practical,” *arXiv preprint arXiv:1702.07521*, 2017.

[19] J. V. Cleemput, B. Coppens, and B. De Sutter, “Compiler mitigations for time attacks on modern x86 processors,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 23, 2012.

[20] M. Payer, “Hexpads: a platform to detect “stealth” attacks,” in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.

[21] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, “Cacheshield: Protecting legacy processes against cache attacks,” *arXiv preprint arXiv:1709.01795*, 2017.

[22] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, “Spydetector: An approach for detecting side-channel attacks at runtime,” *International Journal of Information Security*, pp. 1–30, 2018.

[23] N. A. Simakov, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, R. L. DeLeon, and T. R. Furlani, “Effect of meltdown and spectre patches on the performance of hpc applications,” *arXiv preprint arXiv:1801.04329*, 2018.

[24] S. Chen, F. Liu, Z. Mi, Y. Zhang, R. B. Lee, H. Chen, and X. Wang, “Leveraging hardware transactional memory for cache side-channel defenses,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018, pp. 601–608.

[25] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, “Ric: Relaxed inclusion caches for mitigating llc side-channel attacks,” in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 7.

[26] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” *arXiv preprint arXiv:1801.01203*, 2018.

[27] Z. Allaf, M. Adda, and A. Gegov, “A comparison study on flush+ reload and prime+ probe attacks on aes using machine learning approaches,” in *UK Workshop on Computational Intelligence*. Springer, 2017, pp. 203–213.

[28] W. Tang and Z. Mi, “Secure and efficient in-hypervisor memory introspection using nested virtualization,” in *Service-Oriented System Engineering (SOSE), 2018 IEEE Symposium on*. IEEE, 2018, pp. 186–191.

[29] T. Zhang, Y. Zhang, and R. B. Lee, “Cloudradar: A real-time side-channel attack detection system in clouds,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.

[30] X. Wang and R. Karri, “Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters,” in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 79.

[31] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, “Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks,” 2017, https://eprint.iacr.org/2017/564.

[32] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” Cryptology ePrint Archive, Report 2015/1034, Tech. Rep., 2015.

[33] J. Nomani and J. Szefer, “Predicting program phases and defending against side-channel attacks using hardware performance counters,” in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2015, p. 9.

[34] Z. Zhang and J. M. Chang, “A cool scheduler for multi-core systems exploiting program phases,” *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1061–1073, 2014.

[35] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, “Discovering and exploiting program phases,” *IEEE micro*, vol. 23, no. 6, pp. 84–93, 2003.

[36] G. L. Valentini, W. Lassonde, S. U. Khan, N. Min-Allah, S. A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej *et al.*, “An overview of energy efficiency techniques in cluster computing systems,” *Cluster Computing*, vol. 16, no. 1, pp. 3–15, 2013.

[37] A. Skrenes and C. Williamson, “Experimental calibration and validation of a speed scaling simulator,” in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 2016, pp. 105–114.

[38] J. Lau, S. Schoenmackers, and B. Calder, “Transition phase classification and prediction,” in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 278–289.

[39] A. S. Dhodapkar and J. E. Smith, “Comparing program phase detection techniques,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 217.

[40] C. Ding, S. Dwarkadas, M. C. Huang, K. Shen, and J. B. Carter, “Program phase detection and exploitation,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 8–pp.

[41] Z. Allaf, M. Adda, and A. Gegov, “Confmvm: A hardware-assisted model to confine malicious vms,” in *2018 UKSim-AMSS 20th International Conference on Computer Modelling and Simulation (UKSim)*. IEEE, 2018, pp. 49–54.

[42] L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[43] ——, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[44] V. Nikulin, G. J. McLachlan, and S. K. Ng, “Ensemble approach for the classification of imbalanced data,” in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2009, pp. 291–300.