SpyCatcher: Lightweight Online Approaches for Detecting Cache-Based Side Channel Attacks

Yusuf Külah, B.S.

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University

January, 2015

APPROVED BY:

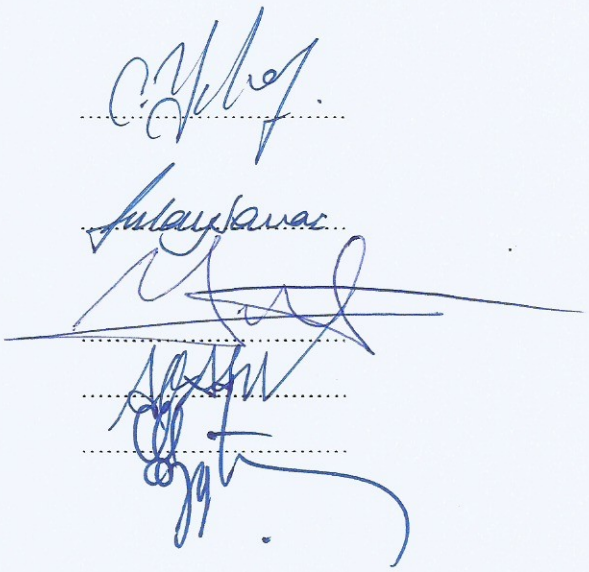
Assist.Prof.Dr. Cemal Yılmaz .....
(Thesis Supervisor)

Prof. Dr. Erkay Savaş .....
(Thesis Supervisor)

Assoc.Prof.Dr. Yücel Saygın .....

Assoc.Prof.Dr. Albert Levi .....

Assoc.Prof.Dr. Özgür Erçetin .....



DATE OF APPROVAL: .....

© Yusuf Külah 2015
All Rights Reserved

SpyCatcher: Lightweight Online Approaches for Detecting Cache-Based Side Channel Attacks

Yusuf Külah, M.Sc.

Computer Science and Engineering, Master's Thesis, 2015

Thesis Supervisor: Cemal Yılmaz

Thesis Supervisor: Erkay Savaş

Abstract

With the increasing complexity of cryptographic algorithms, attackers are looking for side channels to compromise private data. While attackers are tracking side channels, they leave traces behind them unintentionally. In this work, we concentrated on Flush+Reload type of attacks which is aimed to retrieve private data by using intentional contentions on shared resource. Our shared resource is L1 Data Cache of CPU. The trace of attackers on shared resource is a great asset for extraction of utilization pattern which is strong indicator for presence of attacker in the system. For this reason we collected data and extract utilization characteristics of the resource by using hardware performance counters. In this work, by taking the advantage of machine learning approaches, we make a decision on running applications, whether attacker application is one of them or not. Smarter attackers may flush cache partially in order to minimize footprint on shared resource. Workload level is another significant factor that alters the utilization profile of shared resource. For this reason, we experimented our approaches under 4 different levels of partial cache flush and 7 different workload level which mimics e-commerce server load. Our approach is able to detect the presence of attacker with higher than 85% accuracy and lower than 0.5% average execution time overhead.

Önbellek Tabanlı Yan Kanal Saldırılarının Tespiti için Hafif Çevrimiçi Yaklaşımlar

Yusuf Külah

Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans, 2015

Thesis Supervisor: Cemal Yılmaz

Thesis Supervisor: Erkay Savaş

Özet

Kriptografik uygulamalardaki karmaşıklık arttıkça, atak yapmak isten kişiler, hedefledikleri veriye ulaşmak için yan kanalları kullanmaya başvururlar. Ancak yan kanalları takip ederlerken, arkalarında farkedemedikleri izler bırakıyor olabilirler. Bu tezde, önbellek tabanlı yan kanal ataklarından olan Flush+Reload tipindeki atakları yakalayabilmek için çalışmalar yaptık. Bu tip ataklar, ortak aygıtlar üzerindeki kullanım tiplerinden gizli bilgileri elde etmeye çalışırlar. Bu tezdeki ortak aygıt, işlemcinin 1. Seviye önbelleğidir. Casusların önbellek üzerinde bıraktıklar izler, onların varlığını bulabilmek adına oldukça değerli bir bilgidir. Bu sebeple seçilen ortak aygıt üzerindeki kullanım ayrıntılarını donanım performans sayaçları ile topladık. Bu tezde otomatik öğrenme yöntemlerini kullanarak, casus programın sistemde çalışıp çalışmadığını buluyoruz. Arkalarında daha az iz bırakmak isteyen casus yazılımlar, önbellğin yarısını veya daha küçük bir kısmını silerek, bellek üzerindeki izlerini küçültmeye calışabilir. Önbellek kullanımı üzerinde önemli etkisi olan bir diğer parametre de sistem üzerindeki iş yüküdür. Bu sebeple, bu tezdeki yaklaşımlarımızı 7 farklı iş yükü altınca ve 4 farklı silme stratejilerini kullanarak denedik. Bu tezde açıklanan sistem, casus yazılımın varlığını %85 den daha da yüksek bir oran ile bulurken çalışma süresine gelen ek yük, ortalama %0.5 seviyelerindedir.

to my beloved family...

Acknowledgements

This thesis would not have been possible without the support of my supervisors, committee, friends and family. First of all, I would like to express my deepest gratitude to my thesis supervisors Assist. Prof. Dr. Cemal Yılmaz and Prof. Dr. Erkay Savaş. They made valuable contribution to my thesis work with the help of their ideas, immense knowledge as well as their guidance and encouragement. I also would like to thank to my thesis jury, Assoc. Prof. Dr. Albert Levi, Assoc. Prof. Dr. Yücel Saygın and Assoc. Prof. Dr. Özgür Erçetin for their valuable suggestions and inquiries.

I would like to express my gratitude to Berkay Dinçer for his contributions to my thesis study who is a M.Sc. student in Sabanci University. At the beginning, he started to work in this project as a part of course project. He contributed to this work by offering to use supervised machine learning techniques for spy detection.

I would like to thank to my friends Candemir Döğer, Orçun Orhan, Oral Hurhun and members of Cryptography and Information Security Laboratory for their ideas and motivation. Also, I would like to thank to İnci Sarı who has a special place in my heart, for her endless support, motivation and positiveness.

Last but not least, I want to express my special appreciation and thanks to my parents and my sisters as they have always supported and encouraged me. I would not be here without the unlimited love and support they provided throughout my life. Especially, I would like to thank to my elder sister Berrak Külah for abandoning her room and to my sister Işıl Külah for her superior cooking abilities. I am always proud of being a part of my family.

Contents

List of Figures

List of Tables

1 Introduction

Cryptographic algorithms that withstand known theoretical attacks may succumb to side-channel attacks due to flaws in their implementations [24]. Side-channels are unintended manifestations about the key dependent aspects of cryptographic application executions, e.g., the execution time, power consumption, electromagnetic emanation, micro-architectural artifacts, etc. Since the secret key effectively influences the execution of cryptographic applications, observations made on a side-channel may eventually leak information about the secret key if it effects in the computation are not cloaked.

An important category of side-channel attacks is due to shared micro-architectural resources such as cache memory and branch prediction unit [9, 25, 29]. Cache-based side channel attacks, which are the main focus of this thesis, exploit the key-dependent cache access patterns of cryptographic applications. These attacks typically leverage execution times and/or cache contentions to infer whether a cache access is a hit or a miss as well as the actual cache set being accessed. These access patterns are then associated with likely key values to extract the secret key or to reduce the possible key space.

In many cache-based side channel attacks, the malicious party leverages a *spy process* to intentionally create cache contentions with the cryptographic application [38]. One way to develop a spy process is to use a two-step approach, called the *prime and probe* approach [38]. In the prime step, the spy process fills the cache (e.g., L1 data cache memory) with its own data. In the probe step, the spy accesses the cache sets that were previously filled with its own data. Each access is timed. The spy process carries out these steps one after another in a loop. When the spy accesses a cache set in the probe step, since the cache set was filled with the spy's own data in the prime

step, the respective data is expected to be fetched from the cache memory. Thus, the access should be performed fast. However, if an access to a cache set takes longer than expected, then it indicates that one or more cache lines in the respective cache set are evicted by other processes. In the prime and probe approach, such evictions are assumed to be caused by the cryptographic application, although this may not be necessarily the case, as any other process currently running on the same platform can cause these evictions. To factor out the noise, the spy process performs a series of observations, i.e., the prime-and-probe loop iterates a number of times. In effect, the spy process searches for cache contentions. The contentions are then associated with likely key values on a per key byte basis to extract the secret key bytes or to reduce the space of possible key candidates [29,38]. The results of many empirical studies strongly suggest that the same and similar attacks can effectively and efficiently extract private keys [43, 44, 44].

The countermeasures against these attacks are often taken in a retroactive manner [34,38,41,46]. First, an attack surfaces itself. Then, the attack is analyzed and the necessary countermeasures to prevent future attacks of the same or similar type, are determined. Finally, the countermeasures are implemented and the cryptographic applications are redeployed. However, spy processes have been quite successfully adapted themselves to these countermeasures [34,38].

In this thesis, rather than proposing yet another retroactive countermeasure, we present an online approach to detect ongoing attacks, so that proactive preventive measures can be taken in time. The roots of this approach stem from a number of simple observations we make on the common characteristics of all side-channel attacks carried out against software implementation of cryptographic algorithms. First, the victim and the spy share at least one resource, such as data cache, instruction cache, dynamically linked libraries, etc. Second, the spy carries out the attack by creating intentional contentions in the shared resource, whose affects on the victim can easily be observed from outside the victim. Third, since the shared resource is often used by other processes running concurrently with the victim, the spy effects not only the victim but also the other processes. Fourth, the spy is forced to conduct a series of

experiments in order to factor out the noise caused by the other processes. These observations suggest that there are identifiable and repeatable patterns in the behavior of spy processes and that similarities and deviations from these patterns are highly correlated with the presence or absence of spy processes.

To test this hypothesis we have developed an online approach to detect the presence of spy processes. Given a shared resource to be monitored, this approach continuously monitors the resource at runtime and quantifies the contentions experienced by the victim process as well as by the other processes. Whenever the extent to which processes suffer from these contentions reaches a "suspicious" level, a warning is issued indicating the presence of a likely spy process. Once a warning is issued, proactive countermeasures to prevent the ongoing attack, such as migrating the victim to a different core, CPU, or a machine, can be taken. However, such countermeasures are beyond the scope of this work.

We have also conducted a set of experiments to evaluate the proposed approach. In these experiments, while we used the approach to detect the presence of a specific type of spy processes, the techniques are equally applicable to detect other spy processes that operate by creating intentional contentions in shared resources. In particular, the spy process in the experiments, implemented the prime-and-probe approach [38]. The victim process was a cryptographic application using 128-bit AES for encryption and decryption. The shared resource we monitored was L1 data cache memory. The contentions were quantified by using hardware performance counters – CPU resident counters that record various low level events occurring on a CPU. To analyze the contentions, we have developed two approaches: a supervised and an unsupervised approach. The supervised approach assumed that the capabilities of the spy process was known a priori, whereas the unsupervised approach did not have this assumption. In both approaches, the monitoring was carried out on a per scheduling quantum basis by using dynamic operating system kernel instrumentation. Furthermore, the spy and the victim processes were executed on a platform that mimicked an e-commerce server. To analyze the sensitivity of the approach to workload levels, 7 different levels of workloads were used. To evaluate the proposed approach on smarter spy processes, the propor-

tion of the cache memory flushed by the spy process was varied from full, to half, to quarter, to one eighth. We measured both the accuracy and the runtime overhead of the proposed approach.

With all threats to external validity in mind we believe that our study supports our basic hypothesis that there are identifiable and repeatable patterns in the behavior of spy processes and that these patterns can be analyzed at runtime to detect the presence of spy processes, all at acceptable runtime overheads. We arrived at this conclusion by observing that the proposed approach correctly detected the presence of the spy processes with an F-measure of above 85% and a runtime overhead of below 1.5% and that the performance varied marginally over different levels of workloads and the proportions of the cache memory flushed, or whether a supervised or a unsupervised approach was used.

The remainder of the thesis is organized as follows: Section 2 discusses the related work; Section 3 provides background information on various technologies used; Section 4 discusses the proposed approach in a nutshell; Section 5 presents the details; Section 6 discusses the implementation details; Section 7 empirically evaluates the approach; and Section 8 discusses threats to validity; and Section 9 presents concluding remarks and future work.

2 Related Work

In this thesis, we aim to provide a lightweight mechanism for detection of processes that create collusions with a cryptographic process in the cache memory resulting in cache-based side-channel attacks [22, 29, 31, 39]. While collusions created by any process can give rise to cache attacks, majority of the cache attacks rely on a specifically designed process, which is often referred as *spy* process, targeting fine-grained collusions with the cryptographic *victim* process. One particular cache attack proposed by Bernstein [14] does not rely on collusions by a spy process, but on the unintended collusions by operating system processes as shown in [12, 26, 27]. In this work, however, we only deal with methods for detecting spy processes via the instrumentation of hardware performance counters, deployed in many modern day processors.

Side-channel attacks utilize unintended information about sensitive data (e.g., secret keys) that is released during the execution of a cryptographic algorithm [5, 8, 10, 23]. An observable aspect of the computation such as execution time, power consumption during particular operations, cache access patterns, can lead to side-channel attacks, if it is a function of secret key used during computations. A cache access operation based on a round key byte in a block cipher, a branch instruction whose outcome depends on a bit of a private key in public key cryptography algorithm, execution time that varies with the message and the secret key all lead to side-channel attacks.

Cache-based side channel attacks (or briefly cache attacks) utilize key-dependent cache access patterns either in data [4, 7, 14, 15, 16, 17, 27, 27, 29, 33, 37, 39] or instruction caches [2, 3, 6]. Except for [14], all attacks utilize (or assume the presence of) a spy process that flushes the cache memory before the scheduling of the cryptographic process to create cache collusions with the victim process. Recent works such as [11, 13, 44]

apply fine-grained cache flush operations that evict particular cache lines that are likely to be used by the victim process.

The cache attacks can be grouped into three different categories, namely *a*) access-driven attacks, *b*) trace-driven attacks, and *c*) time driven attacks. In access-driven attacks, the attacker observes accessed or unaccessed cache lines. By using the observation, the attacker infers the private key of the cryptographic system. Tromer *et al.* propose an approach that uses a spy process to identify the unaccessed cache sets/lines by the cryptographic application [38]. Tromer *et al.* present two variants of the attack, namely synchronous and asynchronous attacks. In the synchronous variant, the spy process has the ability to start an encryption process. The synchronous attack can recover a 128-bit AES key after about 300 encryption operations. On the other hand, the asynchronous variant does not have the ability to start an encryption operation at will and therefore the attacker only observes the cache usage for a certain period of time. In the asynchronous attack, 45.7 bit of a 128-bit AES key can be recovered by using cache access pattern data observed in one minute. In early examples of access-driven attacks [38], it is assumed that spy and cryptographic victim processes are scheduled in the same core in a multicore processor. Falkner *et al.* propose the so-called Flush+Reload attack, in which the spy process does not necessarily run on the same CPU-core as the victim process [44]. By taking advantage of an Assembly instruction in Intel processors with X86 architecture, the attacker is able to evict cache lines selectively from all cache levels. This forces all processes to load the cache lines from the memory if they need them independent of the core on which they are scheduled. A miss in all levels of cache memory results in significant data access times that can be utilized as a side-channel.

In trace-driven cache attacks [29], the attacker tries to obtain cache miss and hit patterns (a *trace*) in a series of cache accesses such as the lookup table access during the last round of AES. The observation of the power consumption or electro-magnetic emissions of cryptographic device running a cryptographic algorithm can be used to obtain cache traces assuming that the device is under the control of an attacker. Page shows that a 56-bit DES key provides only 32-bit key security if trace-driven attack is

applied [29].

Time-driven attacks [39] utilize the variation in the execution time of the cryptographic algorithm. The attacker tries to distinguish the execution time variations due to cache misses and guess whether a particular cache access of the cryptographic process is a miss or hit. Knowing either the plaintext or the ciphertext, the attacker can produces hypothesis about the outcome of a particular cache access that is determined by the value of a round key byte. If the hypothesis is correct, the observed execution time will coincide with the time model derived from the hypothesis and this will provide the likelihood score of a key value for its being the correct key. The process will yield the correct key with the highest score if sufficient number of observation can be done with certain accuracy.

In order to prevent side channel attacks, many countermeasures are proposed in the literature. An obvious solution is that processes do not share hardware resources in the computer [32], [35]. In a more sophisticated defense method proposed in [41], the authors propose a scheduler based defense mechanism against the Prime+Probe attack [38]. In the Prime+Probe attack, the attacker tries to schedule itself immediately before and after the cryptographic application, and also wants the cryptographic process to perform one encryption during one quanta, which is the duration a scheduled process runs before another process is scheduled. Extending the quanta duration will have a negative impact on the attack, since the cryptographic process, then, will have time to perform more than one encryption. In that case, the cryptographic process will be more likely to access all cache lines and sets holding data and lookup tables used in the cryptographic algorithm. That all cache lines are accessed give no information, and therefore the observation will be useless from attacker's perspective.

Since it has been shown that the side channel attacks are applicable in cloud systems, where virtualization and multi-tenancy are norms [34, 46], research in countermeasures specific to cloud systems attracts some attention. In [30], Pattuk *et al.* propose using a threshold scheme for key distribution in cloud to prevent side channel attacks. Thus, each client do not have to save the actual key for encryption operation. For execution of the encryption operation, shared keys are gathered, then execution is completed.

Hardware performance counters are hardware registers that can be used in various contexts. Note that some care needs to be takes as pointed out by McKee *et al.* in [42]. In [45], Porter and Yilmaz used the advantage of HPCs for classification of program executions. Since hardware performance counters are fast for reading and accurate for measurements, they monitor program executions. Another work, that exploits HPCs is [18] where Cohen *et al.* take the advantage of HPCs to profile system in parallel programs and aimed to detect anomalies during the execution. In [21], they also used HPCs and apply profiling concept into cloud platforms. Their aim is efficient resource allocation and scheduling, which is a critical issue in cloud systems.

Also, HPCs are used in to accelerate cache attacks [40]. In [12], HPCs are used to find out the types of cache collusions leading to the attack in [14]. In this work, we also use hardware performance counters to perform accurate measurements of cache miss and/or hit rates incurred by a process. The changes in or patterns of cache misses suffered by a victim process will be utilized to detect the presence of a spy process, which enables cache attacks.

3 Background Information

In this section we provide background information about the approaches and tools used in the thesis.

3.1 Supervised/Unsupervised Learning

Supervised learning is an aspect of machine learning which is used to build a model in order to determine relationships between input attributes and the output attributes. Constructed models can be used to predict the outcome of a model by evaluating the inputs. These constructed models are then used to reveal important knowledge that is hidden in the data set or they can be simply used to predict a label given the inputs.

Unsupervised learning is another machine learning technique that deals with unlabelled data. This type of learning aims to form clusters based on the attributes of the elements. Distinguishing between different classes without training the classifier is a harder problem to solve rather than supervised learning. Some example algorithms are K-means, hidden Markov models, PCA.

In this work both supervised and unsupervised learning methods were used in order to detect cache based side channel attacks. As the supervised method, we used classification trees with data gathered from the L1 cache of the CPU for both identification of workload level and detection of presence of spy process in order to increase the accuracy of our approach. As an unsupervised learning method we used clustering with thresholds in order to find a cluster where spy process do not exists. Afterwards, our approach is able to make decision on the presence of the spy.

3.2 System Tap

SystemTap [36] is a free software for gathering information from running Linux systems. The software has several kinds of probes for monitoring such as user space probes, system call probes, timer probes etc. The key feature of systemTap is instrument both user level and kernel level at the runtime. Therefore execution information can be collected without modifying, compiling, and redeploying the kernel source code. Run time instrumentation mainly used for profiling.

3.3 Prime+Probe

Prime+probe is a timing based attack model of the spy process that is first proposed in [38]. Prime+probe is a two step approach. In the prime step, the spy process fills the cache with its own data. To this end, we used double linked list structure as shown in Figure 1 which has the same size as L1 Data-cache. In the probe step, the spy accesses the cache sets that were previously filled with its own data in the prime step. Each access is timed. The accesses that take longer than the average access time are identified and the respective cache sets are marked as potentially accessed by the cryptographic application.

The rationale behind the Prime+Probe is simple, the spy carries out the prime and probe steps one after another. When the spy access a cache set in the probe step, since the cache was filled with the spy data in the prime step, the respective data is expected to be fetched from the cache. Thus the access should be performed fast. However, if an access to a cache set takes longer than expected, then it can indicate that one or more cache lines in the respective cache set are evicted by another process. In the Prime+Probe approach, such evictions are assumed to be caused by the cryptographic application.

Our attacker model employs the Prime+Probe approach. The Prime+Probe algorithm has strong impact on cache-miss events by creating cache contentions between processes. Number of cache-miss events are strongly correlated with the presence of an attacker application.

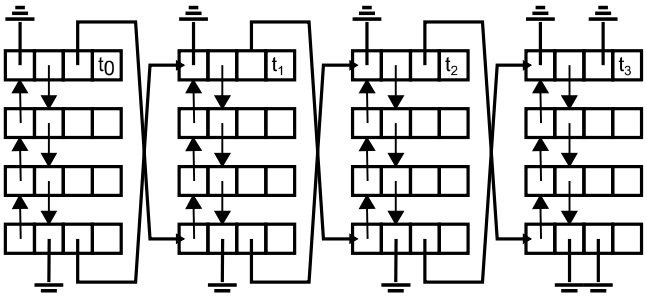


Figure 1: Prime+Probe Data Structure

3.4 Hardware Performance Counters

HPCs are special type of registers inside of the CPU which are dedicated to measure performance of the system. The user can configure counters programmatically, then start to read values. By default, Hardware Performance Counters are deactivated. The counters has special configuration for events to count such as the number of branches taken, number of executed instructions, the number of cache miss and cache access etc. The counter is incremented when specified event occurs in the system. Since the counters are dedicated hardware, access and calculation can be done with low timing overhead. The user programmatically enable/disable the performance counter hardware. Most of the modern CPUs are shipped with sufficient number of HPCs. Observable events on the system via HPCs may differ between CPUs.

3.5 Faban and Olio

Faban framework [19] is an open-source performance testing tool. Users can develop their own workload to test their system's performance. In our case, we used Apache Olio project [20], which is a sample application to help developers evaluate their performance using Faban framework. We created realistic server workload that consists of database and webserver requests. Olio is basically simulates users, and number of concurrent users is configurable over Faban framework.

In this thesis, we used Faban and Olio to generate realistic e-commerce server workload with 7 different levels. Since it is an e-commerce server, workload level is changed by number of concurrent users which is a reconfigurable parameter in Olio over Faban

framework. We have minimum 25 concurrent users and maximum 500 concurrent user on the e-commerce server.

The rationale behind the workload generation idea is the following, the workload level on the server is strongly correlated with the number of cache misses and accesses that cryptographic application and other processes suffers. For this reason, we created 7 different levels of workloads on the server. In Figure 12, the difference of workload levels in the manner of cache usage and cache-miss counts is displayed.

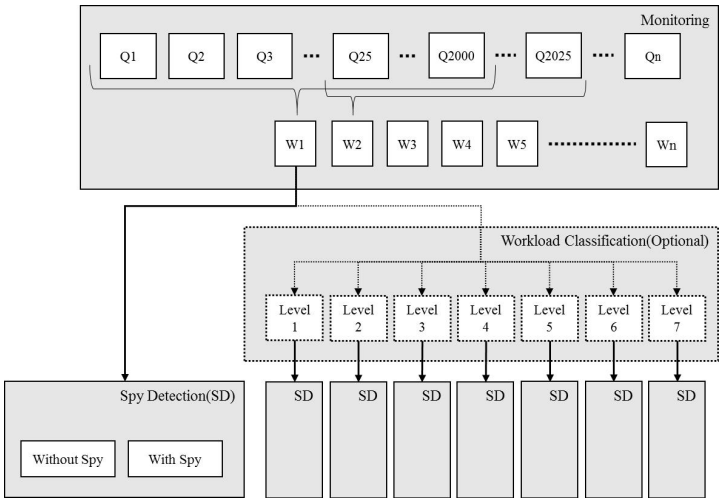


Figure 2: Overview of proposed solution

4 SpyCatcher in a Nutshell

In this section, we will describe the proposed approach in a nutshell. SpyCatcher has 2 modes of operation which are with and without workload classification. At a high level SpyCatcher consists of 3 building modules as it is shown in Figure 2. First we monitor system and collect data. If mode of operation includes workload classification, it is performed at the second step. In the last step, spy detection mechanism works. Task of the monitoring module is to collect values of HPCs for each quanta which is described in Figure 4. After sufficient amount of data is collected, we create sliding window structure as it is described in Figure 5. If the mode of operation is with workload classification, as the next step we determine the workload level of window by using a supervised method. According to the workload level decision, parameters of spy detection mechanism is tuned and run on current window. Our analysis method is the last module before the presence of spy is decided. We have two different methods for spy detection module, supervised and unsupervised methods. After spy detection module

make a decision about the presence of spy, One cycle of proposed solution accomplished. Then our system collect data for required amount of quanta to create new window and whole process starts over.

5 Approach

In this section, we describe four methods to detect cache based side channel attacks by taking the advantage of some machine learning algorithms. Our first approach employs supervised method for spy detection without workload classification. The second one also uses supervised method for spy detection, but it classifies workload levels and create model for each workload level. The third approach is unsupervised, which creates a model for the safe state of the system in order to detect presence of spy. The last approach is a hybrid of supervised and unsupervised algorithms. This approach employs workload classification as supervised approach, and spy detection as unsupervised approach.

Workload classification is done only using supervised approach with total of 10 features were used which are listed in table 1. Supervised spy detection also uses these 10 features. For unsupervised spy detection module, relatively simple feature set is used compared to workload classification phase. Features of spy detection module is listed in table 2.

In order to create training set for the proposed technique, we divided collected data 70% as training set and 30% as the test set in stratified manner. We used same set in order to train both supervised approaches and unsupervised approaches. We tested both approaches with the same test set, which are different than the training set.

As it is described in the Section 4, our proposed solution consists of 3 main modules. In the subsections of this section, we will explain these modules in details which are monitoring, workload classification and spy detection.

ID	Description
1	Mean of miss rate
2	Median of miss rate
3	Mean-standard deviation of miss rate
4	Mean+standard deviation of miss rate
5	Average miss count
6	Average access count
7	Number of unique processes in a window size n
8	Average number of unique processes in sub-window, n/2
9	Average number of unique processes in sub-window, n/4
10	Average number of unique processes in sub-window, n/8

Table 1: Features for Supervised Approaches

ID	Description
1	Mean of miss rate
2	Mean-standard deviation of miss rate
3	Mean+standard deviation of miss rate

Table 2: Features for Unsupervised Approaches

5.1 Monitoring

In this section, we will describe the data collection step. Data collection is done by taking the advantage of SystemTap and custom patched Linux kernel where hardware performance counters are reachable without any additional software. We have used two hardware performance counters to measure required CPU events which are L1 data cache-miss and L1 data cache-access. These values are read for each processes at the beginning of quanta (i.e. when the scheduler takes process into execution) and at the end of quanta. Reading pattern of HPCs is shown in figure 4. In the event cycle of scheduler, kernel functions "`perf_event_task_sched_in`" and "`perf_event_task_sched_out`" are called for scheduling a process in and out, respectively. These kernel functions are instrumented using SystemTap. We have two values, one for the before quanta execution and the other for the after quanta execution. Difference of each counter's values mean the number of occurred event during one quanta in CPU.

As it is shown in figure 4, counter values read at the beginning (M_1, A_1) and at the end (M_2, A_2). For the shown quanta, cache-miss count, cache-access count and cache-miss rate are calculated as follows,

$$Miss(Q_n) = M_{2,n} - M_{1,n} \tag{1}$$

$$Access(Q_n) = A_{2,n} - A_{1,n} \tag{2}$$

$$MissRate(Q_n) = \frac{Miss(Q_n)}{Access(Q_n)} \tag{3}$$

After the calculation of these values for each quanta, an analysis window will be formed. We realized that number of quanta that forms an analysis window is significant for performance of the system. In order to decide on a window size, we conducted an experiment. We have two variables for this experiment, window size and shift amount. Window size is the number of quanta that forms an analysis window. After an analysis window is formed and processed, shift amount of quanta is required to form a new window for analysis. We measured the accuracy of workload classifier against different

window sizes and different shift amounts. Visual explanation of window and shift amount term can be found in Figure 5. Detailed results of the experiment are shown in Figure 3. The aim of the experiment is to discover a fair size for the analysis window, not the fully optimized size. Discovery of an optimized window size is out of the scope of this thesis.

As a result of the experiment, we decided to select window size as 2000 and shift amount as 25. but the number of quanta that by gathering features of 2000 quanta as described in figure 5.

$$W = w_1, w_2, w_3.....w_N \tag{4}$$

$$w_i = \sum_{y=0}^{n} q_y \tag{5}$$

where W is a set of windows describing the execution, w_i is the current window in execution q represents quanta belonging to execution. We calculated set of features for each analysis windows, which mainly consists of descriptive statistics, as listed in Table 1.

We collected data, under seven different workload levels. We have used Faban and Olio to generate realistic server workload. We defined number of concurrent users for workload level 1 to 7, as respectively 25, 50, 100, 200, 300, 400 and 500. Increasing the number of concurrent users results increase in the workload level.

5.2 Workload Classification

Data analysis among different workload levels suggests that workload level on the environment effects cache behavior. We noticed an increasing trend in miss rate when workload was high in the environment. This information, naturally, effects our decision whether there exists a spy in a given window or not.

Windows that are a part of different workload levels will have different cache usage characteristics even though they consist of same processes as it is described in Section 7.5. A window that is executed when the number of concurrent users is 500, will certainly suffer more cache miss rate compared to a window when there is 25 concurrent

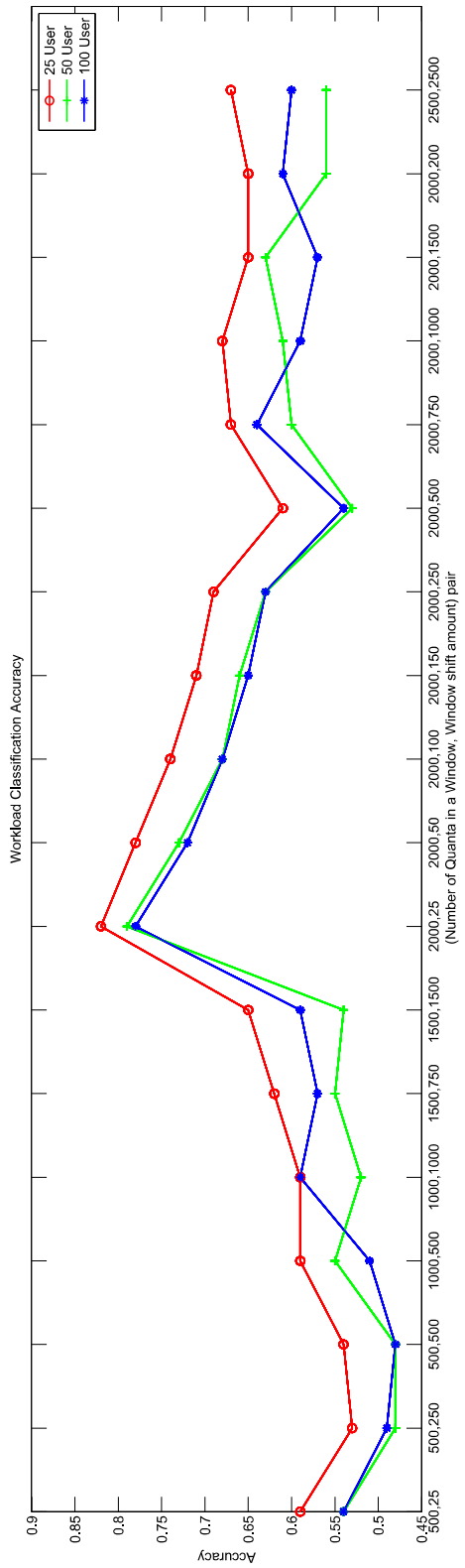


Figure 3: Window Size Decision

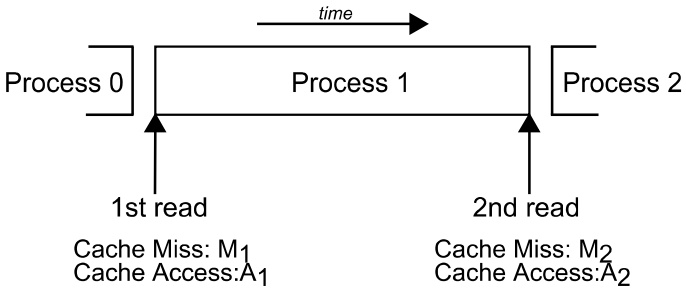


Figure 4: Counter reading pattern

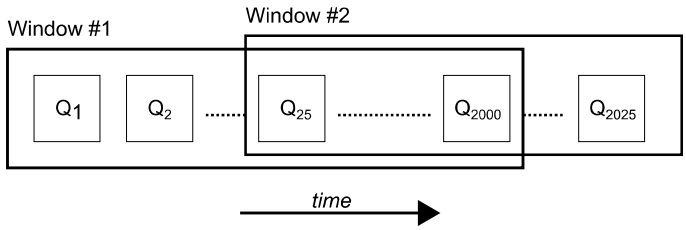


Figure 5: Sliding Window Structure

users as it is shown in Figure. Therefore grouping windows by their workloads before making the final decision was a good step in order to increase our accuracy.

Purpose of this step was simple, build a workload classification algorithm so we can train specialized classifiers for making the final decision. We extracted features in Table 1 for workload classification. The workload classifier is a decision tree, build with C4.5 algorithm, we aimed to minimize the entropy while building the tree. Since C4.5 is a supervised algorithm, we need to train it first, for this reason we divided the whole dataset into 70% training set, and 30 % test set. We have 7 different workload labels in our dataset, so C4.5 algorithm creates a 7 class classifier. We tested the decision tree by using our test data in order to measure accuracy. After training phase of classifier, we converted it to native C code to embed into the running Linux kernel, which resulted in nested if else conditions. A representation of a workload classifier was given as a Figure 7.

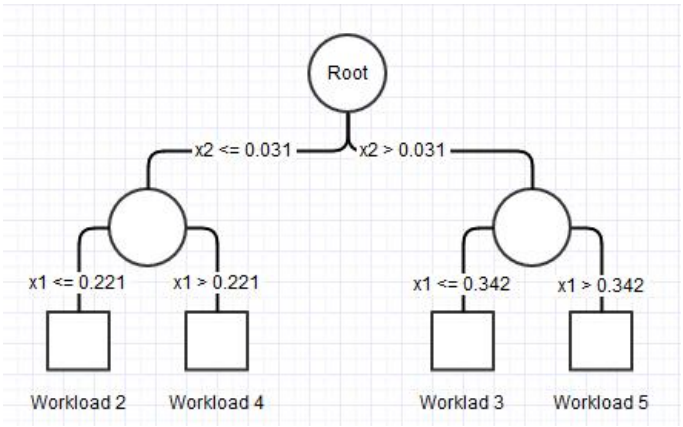


Figure 6: Decision tree for workload classification

```
if(x2 > 0.031){
    if(x1 > 0.342){
        workload_level = 5;
    }
    else{
        workload_level = 3;
    }
}
else{
    if(x1 > 0.221){
        workload_level = 4;
    }
    else{
        workload_level = 2;
    }
}
```

Figure 7: Workload classifier sample code

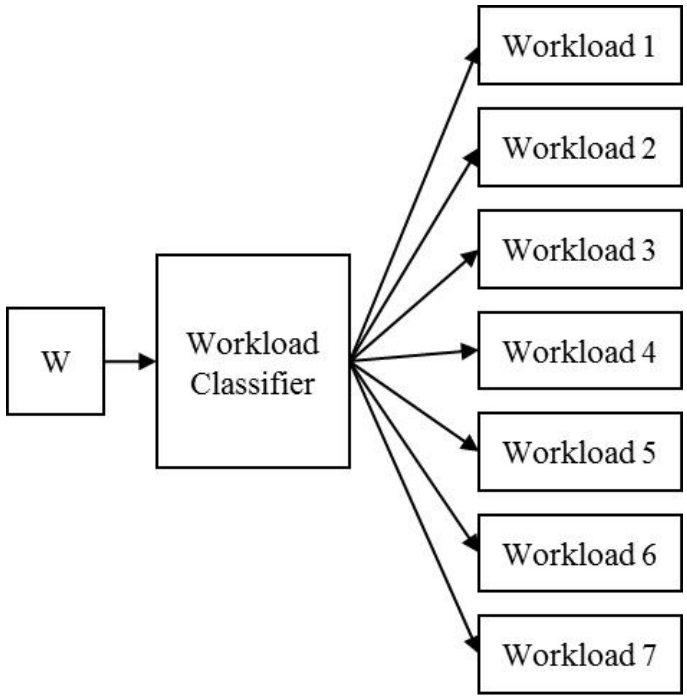


Figure 8: Workload classifier model

5.3 Spy Detection

This section describe the process of spy detection. SpyCatcher have 2 different spy detection mechanisms which are supervised and unsupervised method. Supervised method was done to set an upper bound for the unsupervised method in terms of accuracy. Since supervised method is specifically trained for the spy implementation used in this thesis it is natural to expect a higher accuracy than the supervised method. However specifically training a classifier for a certain way of spy implementation is far from ideal. Any change regarding the spy implementation would cause this specifically trained classifier to fail. Nevertheless we used both supervised and unsupervised method as the detection method of SpyCatcher.

Supervised spy detection method consists of 7 different classifiers which are specialized for each workload. After the initial classification of the Workload classifier, execution data was fed to a specialized decision tree. The training of this decision tree was done by tagging each workload level by looking at the current workload on the system and we tagged the window as with spy if there is a spy in it. We have 1 label

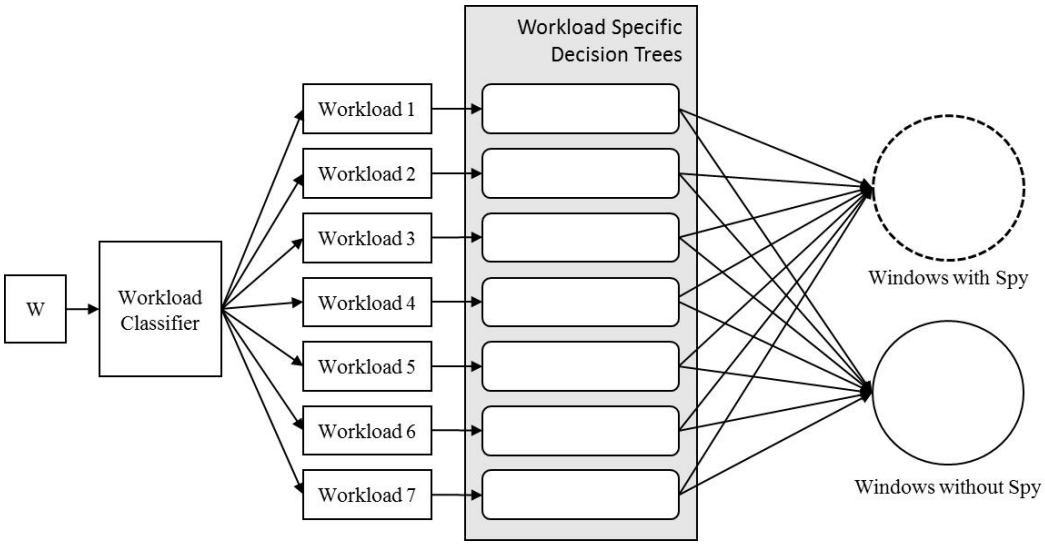


Figure 9: Overview of Supervised Spy Detection

in supervised mode of operation of SpyCatcher which is with spy and without spy.

If SpyCather use workload classifier to detect workload level, it creates a model per workload level for spy detection module as it is shown in Figure 9.

Note that each classifier is also embedded into the kernel and they can be expressed as nested if-else statements written in C language. In principle, spy detection classifiers are similar to Workload classifier which we discussed above. They are both nested if-else statements implemented in C language generated by C4.5 decision tree algorithm.

5.3.1 Supervised Method

In order to understand whether there exists a spy in an analysis window we used supervised machine learning algorithms. Supervised method is one of the solutions for spy detection module in Figure 2.

We conjecture that the presence of a spy process change the cache behaviour. There are a total of 7 classifiers trained according to workload levels, after training the classifiers for all workloads we proceed to feed the data generated by the workload classifier to the corresponding spy detection classifier.

Spy detection classifiers are relatively simple compared to the workload detection classifiers. They result in lighter trees and they use less features. As mentioned above

spy detection classifier consists of 3 features.

We specifically used decision trees as our classification engines. There are several reasons behind this decision. To start with, decision trees are easy to implement in kernel level since a tree can be expressed as a certain number of nested if-else statements. Another reason to use decision trees was to avoid the extra overhead caused by calculations. A naive-bayes classifier or a support vector machine classifier would introduce extra overhead due to some complex arithmetic operations. In the case of decision trees, overhead to execution time is minimal. There are no arithmetic operations to be performed other conditional branch instructions.

5.3.2 Unsupervised Method

The second solution that we propose for spy detection module is an unsupervised method. In this approach we calculate a safe zone where windows do not include spy process. In order to form such a zone, we used windows without spy process in the training set.

The rationale behind the idea is the following, we can track a system for a certain time interval while spy process is not running in the system. We create a model using features of windows to define safe state of the server. In order to create model, we used the features which are listed in Table 2. By reducing the number of features, windows are transformed into a point in 3-Dimensional domain. We need to find the centroid of the points for each workload level. For this reason, we aimed to take advantage of centroid calculation strategy of k-means clustering algorithm in order to calculate suitable centroid point for our data, we k-means algorithm. We feed k-means clustering and request for one cluster, it calculates the centroid of the given data. Once the centroid is calculated, we measured distance of each point's in test data to the centroid using manhattan distance algorithm. Since some of the points are outlier in the dataset, we discard points with the distance larger than the 3rd quartile of the dataset. We observed that, 95th percentile of distance is an acceptable threshold in terms of performance measurements when outliers are discarded.

The threshold value which is set using training data, identifies the safe region of

windows in 3-D space. In Figure 10, distribution of windows are shown in 3-D domain. Green points identifies windows without spy process, red points identifies windows with spy process and the wireframe of sphere identifies the calculated safe region of windows. As it is shown in Figure 10, windows without spy process and with spy process are separable in the domain of selected features. In order to identify different classes in dataset, we used thresholding technique against distance to centroid metric. In Figure 10, we have 2078 window without spy process(green points) and 1962 of them located in the safe region sphere. There is 2071 windows with spy(red points), where only 57 of them located in the safe region, the others are labelled as with spy. Once the model of safe state created, our system is ready to make decision on the presence of the spy process. The first decision is made, when 2000 quanta is collected as the first window. After SpyCatcher decide on the status of the first window, it needs data from 25 new quanta to form a new window. Then we can make a decision about a new window whether it is valid for our model or not. The window(or point in selected domain) considered as without spy process, if it is placed inside of the sphere like green points in Figure 10. Otherwise, that window marked as with spy process.

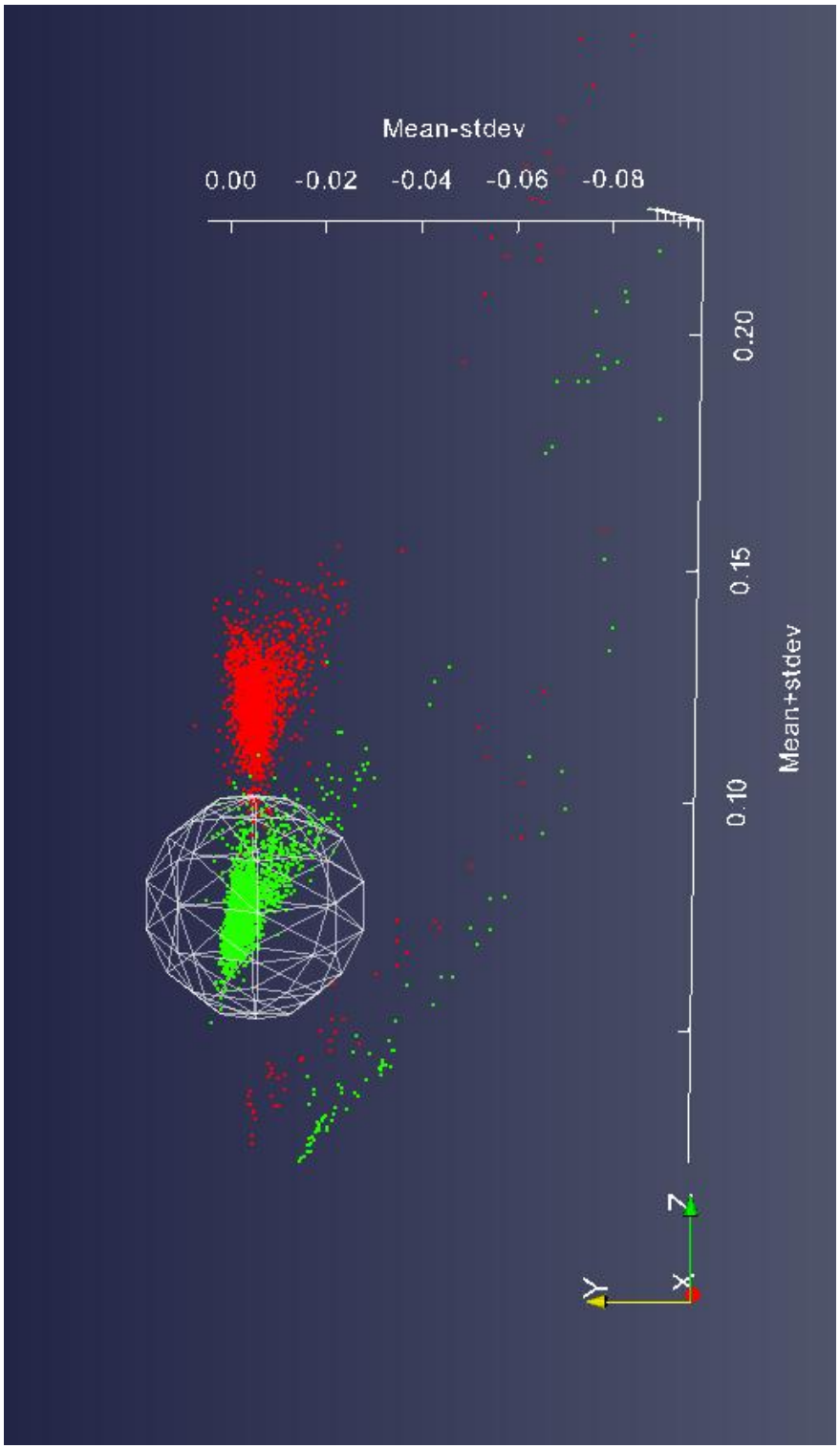


Figure 10: Distribution of Windows for Unsupervised Spy Detection

6 Implementation

In this section, we will mention about implementation details of core elements of this thesis, which are the attacker process and the tracer process.

6.1 Spy Application

The attacker process is designed to evict specified number of cache-sets from L1 data-cache of the CPU. In order to perform eviction, attacker uses an array whose size is same with the cache sets that will be evicted. We have 64 sets in L1 data-cache. The size of attacker's table varies in terms of number of cache-set to flush.

The spy application successfully evicts cache-sets from the L1 data-cache. In order to be sure on flushing the cache, we conducted an experiment. We observed cache-miss count during the execution of 1 AES encryption with pseudo-randomly generated 128-bit key and 128-bit pair. We used openSSL library for AES implementation with version 0.9.7a. Descriptive statistics are displayed in Figure 11. Theoretically, all rounds of AES encryption causes around 10 cache-misses. When all of the cache lines are evicted, theoretical number of cache-misses is around 100. Average values in the figure corresponds to the theoretical calculations. Because of the other executions during the regular job cycle of the CPU such as memory allocation, function calls etc., theoretical values do not exactly met with the observed values.

6.2 Tracer Application

Tracer application is the main component of this thesis which implements the data collection, feature extraction and spy detection logic. This application is developed using

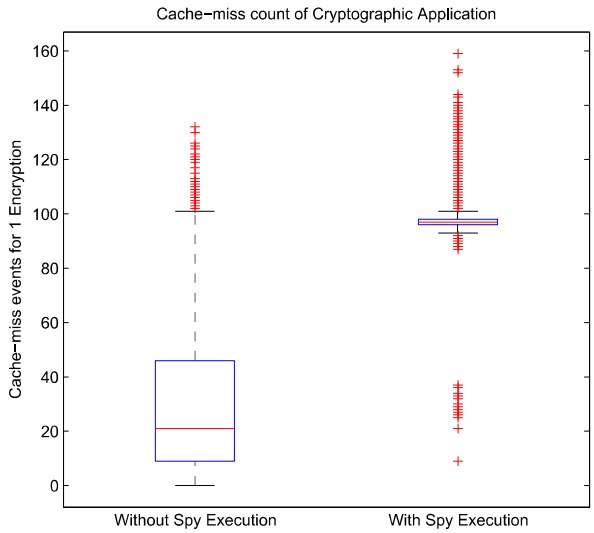


Figure 11: L1 Cache Miss Events

systemTap scripting language and executed in Linux kernel. Data collection section of this application employs 3 arrays which are global variable with size of 2000. We used these array as analysis windows which are described in section 5.

Initial trigger comes from the Linux scheduler to the tracer via systemTap probe that instruments "`perf_event_task_sched_in`" function. When the scheduler takes a process into execution, the value of required HPCs are read and stored in the internal data structure of the Linux process. In regular Linux systems, all processes has its own data structure where meta data of the process is stored such as process ID, thread ID, etc. We added two additional fields for value of HPCs for this thesis to store L1 data cache-miss and L1 data cache-access event counts. When the process execution is finished for a quanta, value of HPCs are read using the same method. When the execution of quanta is finished, we instrument the relevant function which is "`perf_event_task_sched_out`", and execute our whole logic after it is executed. At the end of quanta, we are able to read values from the internal values of process itself. Therefore we have the values of both counters for before quanta execution and after quanta execution. Thus we are able to calculate the L1 cache-miss events and L1 cache-access events for this quanta.

Number of cache-miss and cache-access events are stored in global arrays. At the end

of each quanta, cache-miss, cache-access and cache-miss-rate feature is calculated and stored in global variables. Remaining features in table 1, is calculated when an analysis window filled with required amount of quanta. The calculation details of remaining features will be described in subsections.

Mean of Miss Rate

This feature is calculated by using the miss rate array, which consists of data collected between the context switch time of scheduler. Mean of miss rate for an analysis window is basically the arithmetic mean of the values and calculated as follows;

$$Mean = \frac{1}{2000} \sum_{i=1}^{2000} MissRate_i \tag{6}$$

Median of Miss Rate

Median is the value of the element which is placed in the middle of a sorted array. The miss rate array is not sorted because when values are storing, the challenge is the time. We do not have time to sort array at each context switch. For this reason we sort the miss rate array after data of 2000 quanta collected in kernel level by using quick sort algorithm. Then, select the element in the middle of the array.

Mean + Standard Deviation of Miss Rate

The calculation of mean of miss rate feature is described above. When sum mean and the standard deviation, we find the 3rd quartile of the dataset which helped us to identify differences.

Mean - Standard Deviation of Miss Rate

The calculation of mean of miss rate feature is described above. When subtracting standard deviation from mean , we find the 1st quartile of the dataset which helped us to identify differences.

Average Miss Count

Arithmetic mean of L1 data cache-miss count values for each quanta that are observed in monitoring stage, forms this feature for an analysis window.

Average Access Count

Arithmetic mean of L1 data cache-access count values for each quanta that are observed in monitoring stage, forms this feature for an analysis window.

Number of Unique Processes in an Analysis Window

An analysis window consists of 2000 quanta but the number of unique processes in a window varies according to the workload level. When the workload level on the server increases, basically total number of running processes is likely to increase. We use the relationship between workload level and number of running processes on the system in order to estimate workload level. For this reason we calculated number of unique processes and identify according to the process IDs.

Number of Unique Processes in Partial Analysis Window

Number of unique processes may vary in smaller pieces of the analysis window. For this reason we divided an analysis window into 2, 4 and 8 equal pieces and calculate number of unique processes as it is described above. Then calculate the arithmetic mean of each part's unique number of processes in order to form this feature. This feature is significant for workload classification as well.

After the features are calculated, next step is determined according to the requirements of the experiment. If workload classification is required, then the tracer executes the decision tree which is trained off-line by C4.5 algorithm. Thanks to the C language compatibility, we embedded the decision tree implementation into systemTap script. The last module is the spy detection. In the supervised mode of operation we trained decision tree using C4.5 algorithm offline and executed in kernel level online thanks to the systemTap features. For the unsupervised mode of operation, since this module uses fewer number of features than the supervised spy detection, it consists of several if-else statements at all. This module is also trained offline, but able to make online decision on an analysis window.

7 Experiments

7.1 Subject Applications

We performed experiments to collect performance counter values for two cases; cryptographic application and other applications running on the server.

7.1.1 Cryptographic Application

Cryptographic application performs encryption operation with psuedo-randomly generated message and key where the key size is 128 bit. In every iteration of the application, it generates new message, key pair and then perform encryption. While performing encryption operation, it uses open-SSL implementation of Advanced Encryption Standard (AES) version 0.9.7a.

7.1.2 Other Applications on Server

There are so many running processes in default Linux system. Other Application means all of the processes apart from cryptographic process. In addition to default Linux processes, we added two new operating system service, which are web server and database server. These services are used create realistic user workload on the server by Faban framework.

7.2 Independent Variables

3 independent variables are used in this work, which are;

Traced processes

L1 Data-cache miss and L1 data-cache access values of each process's each quanta

is collected during our experiments. In analysis phase, we divide the collected data into two group. One of the groups includes only cryptographic process's cache-miss and cache-access values. The other group includes all processes in the collected data.

Spy's Cache Flush Percentage

Spy flushes L1 data-cache using Prime+Probe method which was defined in Shamir et.al.'s work [28]. In our work, we flushed 100%, 50%, 25% and 12,5% of L1 data cache of the CPU.

Workload Level

We have 8 different workload levels, which are defined as in the following. First level do not include any workload other than the spy and the cryptographic processes. For other workload levels, different number of concurrent users are simulated using Faban framework. For workload level 2 to 8, we have 25, 50, 100, 200, 300, 400 and 500 concurrent users respectively.

7.3 Evaluation Framework

In the context of our experiments, windows classified as with spy and really includes includes spy evaluated as true positive(TP) samples. Windows classified as with spy but in fact windows with out spy evaluated as false positive(FP) samples. Windows classified as without spy, and really without spy evaluated as true negative(TN) samples. Windows classified as without spy, but in fact windows with spy evaluated as false negative(FN) samples.

In light of these definitions, the performance of our approach is measured using 3 most commonly used metric in performance evaluation which are;

Precision

Precision term can be defined as percentage of really spy including windows in labelled as with spy windows.

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

Recall

Recall is defined as discovery rate of classified as with spy windows among really with spy windows.

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

F-Measure

Only precision or only recall do not really measure the performance of the approach, for this reason f-measure is used which is harmonic mean of precision and recall.

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{9}$$

Runtime Overhead

Our analysis method is running online, for this reason runtime overhead is crucial success factor. We measured runtime overhead using cryptographic process. We measured elapsed time during 100k encryption while both cryptographic process and tracer is running and only cryptographic process is running. Performance metric of Runtime Overhead is calculated by using Equation 10. t_e*(withTrace)* is the elapsed time during while both cryptographic process and tracer process are running. t_e*(withoutTrace)* is the elapsed time during while only cryptographic process is running. We performed the measurement around 10k times. For detection of average time from 10k observation, we divide whole data into smaller chunks, then used the average of averages approach to minimize the noise in data.

$$Overhead = \frac{t_e(withTrace) - t_e(withoutTrace)}{t_e(withoutTrace)} \tag{10}$$

7.4 Operational Model

In this section, we will describe the experiment environment, used version of hardware and software components. We experimented the approach on the server which has Intel Core 2 Duo E6420 CPU, 4 GB of ram, 64 KB of L1 data cache and 1 TB of hard disk drive. The whole approach tested on Centos 6.4 and Linux Kernel version 2.6.32. Kernel

instrumentation is done by taking advantage of System Tap version 2.6. Cryptographic algorithms are used from openSSL library 0.9.7a. For workload generation, we used two tools, version of Faban is 1.2, and version of Olio is 0.2. Handling of workload is done by web server and database server with version 2.2.15 and 5.1.69 respectively.

For training purposes, Matlab 2013b is used both for workload classification and spy detection modules. In the supervised parts of out approach, C 4.5 decision tree algorithm used, which is already implemented in Matlab library.

By definition of Prime+Probe attack, attacker process and target process must run on the same core. For this reason, we used `taskset` command of Linux which forces processes to run on the specified core. Cryptographic process, spy process, web server and database server processes are forced to run on the same core of the CPU.

7.5 Workload Characteristics

Workload is generated on the system using Faban and olio as described in background information section. We used web server and database server workload with different number of concurrent users as workload. Selected side channel attack approach is using cache contentions. In order to detect attacks, different workload levels must have different impact on L1 CPU-cache. Difference of workload levels in the manner of L1 cache-miss event is shown in figure 12.

Number of cache-miss event is increased as the workload level is increased. This means that, higher number of processes is running on the system and these processes are using same cache sets with cryptographic application. In other words, workload levels are distinct in the manner of cache-miss event.

7.6 Results and Discussion

In this section, we will explain and discuss the results of the experiments. 4 different experiments are performed for testing our approach. Half of the experiments do not use workload classification module. Since spy detection methods of experiments are different, we name them accordingly. In supervised method section, spy detection is

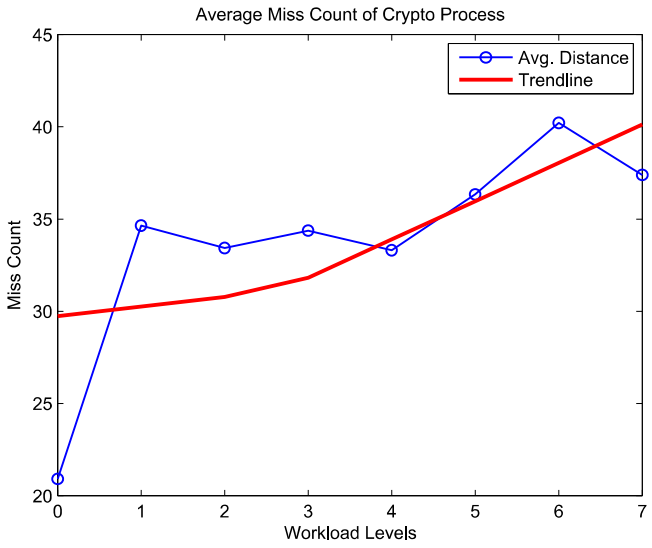


Figure 12: Workload Characteristics

done by supervised machine learning algorithms.

As we stated before, we experimented our approach over collected data both only cryptographic process and all running processes. Performing analysis over only cryptographic application's data is not accurate enough to decide on the presence of the spy process. We conducted series of experiment over only the data of cryptographic application using the same setup. Precision, Recall and F-measures are provided in Appendix B, where overall accuracy is lower than 45% for unsupervised approach and lower than 85% for supervised approaches. For this reason we focused on data of all processes.

7.6.1 Supervised Method

In the case of supervised learning, there were 2 different approaches as explained in previous sections. For the first one, we trained a decision tree for all workloads in order to detect and isolate the spy process. For the second approach we eliminated the Workload Classifier in between.

Two different approaches introduce a time-accuracy tradeoff. If there is a workload classifier in between, we measured F-measure around 99% and runtime overhead as 0.3%. When workload classification is not used, single decision tree is trained for all

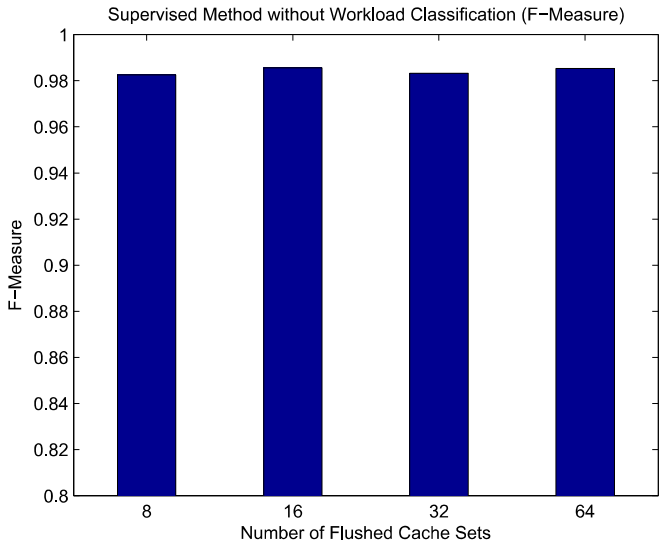


Figure 13: Performance details of Supervised approach without Workload Classification

workload level and we observed F-measure as 98% and runtime overhead as 0.1%. We notice an increase in F-measure since we were able to differentiate and model the workload effect on cache. However, since we use an additional classification tree in between this is causing an extra overhead in terms of timing. Detailed experiment results are provided in Appendix A as Precision, Recall, F-Measure and Runtime Overhead.

Differentiating between workload levels are not mandatory for the approach to work but it is certainly beneficial. Results of this approach indeed proves that the classifier can also perform well without differentiation of workload levels.

Experiments with different configurations were carried out to further investigate the results. Supervised learning, in this research were mostly used to give us an idea about the upper bound in terms of F-measure, precision and recall. Naturally unsupervised learning method is a better approach than supervised method. It is generalizable to every execution without training a specific classifier but it is also intuitive to expect a slightly worse result compared to the supervised approach.

In runtime overhead perspective, overhead introduced by decision trees were negligibly small. We measured 100k encryption with and without spy detection modules in execution. Runtime overhead introduced by spy detection module is less than 1% and

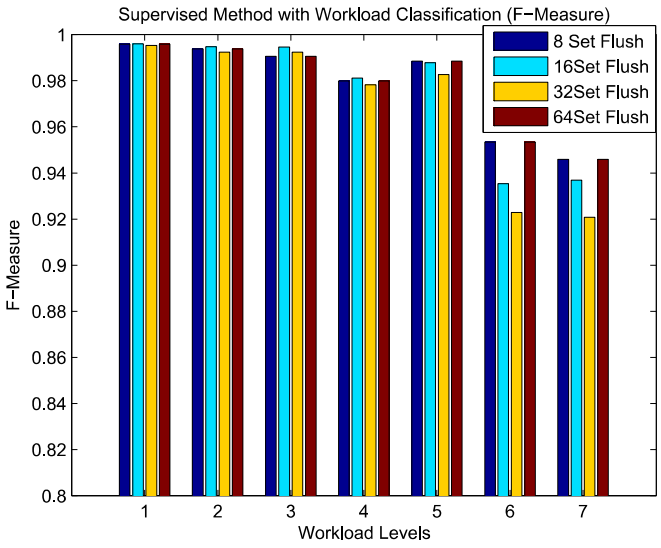


Figure 14: Performance details of Supervised approach with Workload Classification

can be seen in Appendix A.

7.6.2 Unsupervised Method

Two different experiments are performed in the context of unsupervised approach. At first, we used SpyCatcher with workload classification, and we selected unsupervised method for the spy detector module. The model for unsupervised spy detection is formed according to the output of supervised workload classification algorithm. Performance details of the first configuration is shown in Figure 15. As it is shown in Figure 15, F-measure has decreasing trend as the workload level increases. However it is still higher than 85%. The reason behind the negative trend is the growing noise on the system as the workload increases.

The second observation from Figure 15 is that, the success of this decision mechanism is not dependent on the number of flushed cache set. As shown in Figure 15, F-Measures are very close to each other under same workload level. Spy flushes different number of cache sets. Flushed cache set may not belong to cryptographic application, but some other process needs the flushed part of the cache during its execution. This situation leads to cache miss event. Since our system collects data from all of the

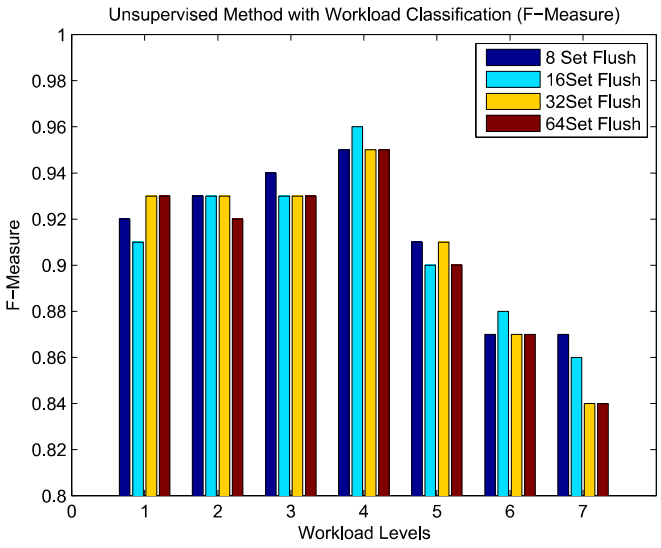


Figure 15: Performance Details of Unsupervised Approach with Workload Classification

running processes, flush operation become detectable by using our approach In fact, partial flush of cache sets creates difference between the processes which are using the flushed sets of the cache and which are not in the manner of cache miss event count and miss rate. Therefore, F-Measure has slightly positive trend as the number of flushed cache set decreased. This result means that, our approach can detect the spy included window when the spy evicts cache sets partially.

The other important aspect of the experiment is runtime overhead of the analysis and decision making progress. Timing overhead details are shown in Appedix A. The overhead measured using cryptographic application. The average runtime overhead was lower than 0.8%.

In the context of unsupervised approach, the second experiment is done without workload classification over the same train and test set. Thus the second configuration employs pure unsupervised mechanisms. This experiment also repeated for 4 different number of flushed cache sets. One centroid and one threshold defined by our proposed method for all workload levels. As the number of flushed cache set increased, average distance of windows to the centroid also increased. Details of distance metric are provided in Figure 17. This indicates us that, when spy process flushes higher number

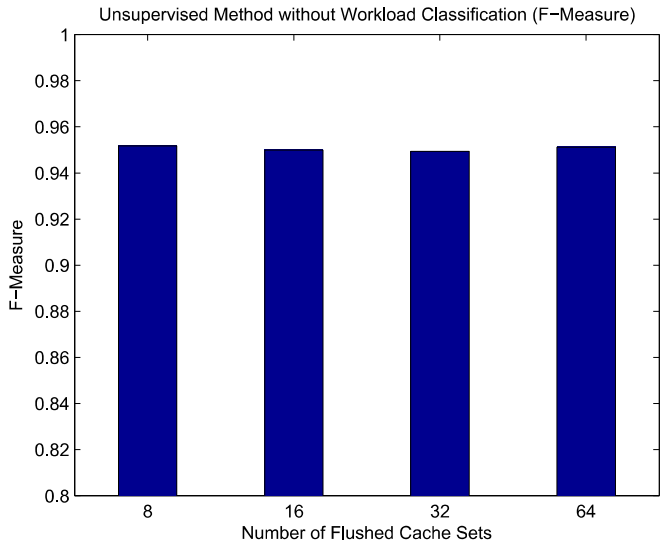


Figure 16: Performance Details of Unsupervised approach without Workload Classification

of cache set, windows become distant from the centroid.

The F-measure result of experiment where SpyCacher executed in unsupervised mode and workload classification is not executed, are provided in Figure 16. As it is seen on the figure, F-measures are around 95% which are higher than the hybrid solution of the problem. The reason is the following, workload classification is not used in this approach, by this way we eliminated the error rate of workload labelling process. By taking out the workload classification module, we are able to build more reliable model for spy detection. Since result of workload classifier is used for building model in hybrid approach, erroneous workload classification led to create unreliable model for spy detection module.

Timing overhead of the method is provided in Appendix A. Overhead values are slightly lower than the other unsupervised configuration. The overhead measured using cryptographic application. The average runtime overhead was lower than 0.1%.

The result indicates that majority of time is spent in workload classification module which is not surprising. By waiving around 5% of F-measure, we can detect the presence of spy processes with lower overhead.

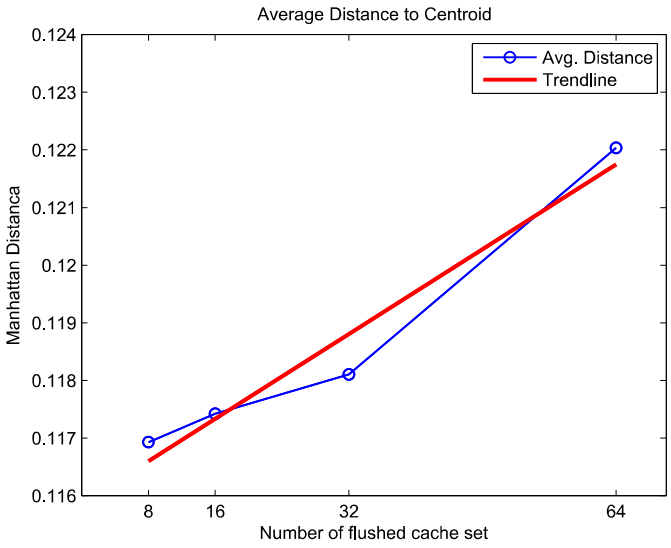


Figure 17: Average distance of windows to Centroid in Unsupervised Approach without Workload Classification

7.7 Compare and Contrast

In this thesis, we have 4 different configuration to evaluate our approach on Linux system. Since supervised approaches more accurate than the unsupervised ones, supervised workload classification and supervised spy detection experiments are done to discover what are the upper limits of F-measure in different number of flushed cache sets. As it is shown in Figure 14 and 13, F-measure is higher than 95% in supervised approaches. Supervised method is successful in the manner of F-measure, but elapsed time for a decision can be reduced by using the proposed unsupervised methods. Using the unsupervised approach reduced the runtime overhead by 90%, compared to using the supervised approach.

8 Threats to Validity

In this section, we will discuss about the details of possible threats to external validity which are *a*) selected environment, *b*) side channel attack model, *c*) number of attacker on the system, and *d*) generated workload. First of all, we only tested our approach in Linux environment which is selected according to our team members' experience. Since programming HPCs is possible in Windows and Mac OS, we strongly believe that our proposed solution can be implemented on other platforms. Although we have only evaluated the proposed approach to detect the presence of spy process using the Prime+Probe technique, we believe that the approach is readily available to detect other spy processes. This is because other spy processes in the cache-based side-channel attack also operate by creating intentional cache contentions, which affect the degree to which the process suffer from cache misses. The other threat is number of spy processes on the system. We experimented our scenarios with one spy process. We believe that detecting the presence of more than one spy process in not any harder than the detecting the presence of a single spy process, because as the number of spy process increase, the cache contentions increased by the processes increase. The last important threat to validity for our work is workload levels. We tried to create workload as close as possible to real life user workload of an web application server. In particular v , we worked with 8 different workload levels. We simulated different number of concurrent users from 25 user to 500 user on our web server and database server. Since our approach based on CPU-core sharing principle, all of the processes need to be forced to a single core of CPU. For this reason, we believe that 500 concurrent user is realistic load for a single core.

9 Conclusion and Future Work

In this thesis, we presented a lightweight online approach to detect the presence of the spy processes. Therefore we collect data from an e-commerce server, then analyze it for a window and make decision on the window whether an attacker exists or not. We located the window with spy process with higher than 85% F-measure and 0.5% runtime overhead in average. Identification of the presence of attacker is critical for overall system security. Once the the presence of attacker is confirmed, applications that are running on the system may take additional actions in order to prevent information leakage.

As the future work of this thesis, suspicious process or a list of suspicious processes can be discovered by using some other technique. The other next step of this thesis can be experimenting our approach using other attacker models. Once the attack is discovered and the attacker is found, then what are the possible actions. Changing the allocated CPU of attacker or terminate the attacker process. Preventive actions are also quite promising.

References

[1] M. Abe, editor. *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *LNCS*. Springer, 2006.

[2] O. Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. *IACR Cryptology ePrint Archive*, 2007:164, 2007.

[3] O. Aciiçmez, B. Bob Brumley, and P. Grabher. New results on instruction cache attacks. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.

[4] O. Aciiçmez and Ç. K. Koç. Trace-driven cache attacks on AES (short paper). In P. Ning, S. Qing, and N. Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.

[5] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. *IACR Cryptology ePrint Archive*, 2006:351, 2006.

[6] O. Aciiçmez and W. Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In T. Malkin, editor, *CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 256–273. Springer, 2008.

[7] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the aes. In Abe [1], pages 271–286.

[8] O. Aciiçmez, J.-P. Seifert, and Ç. K. Koç. Predicting secret keys via branch prediction. *IACR Cryptology ePrint Archive*, 2006:288, 2006.

[9] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Abe [1], pages 225–242.

[10] Onur Aciiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES. *IACR Cryptology ePrint Archive*, 2006:138, 2006.

[11] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Proceedings of Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014.*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[12] Ali Can Atici, Cemal Yilmaz, and Erkay Savas. An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks. In *Seventh International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 18-20 June 2013 - Companion Volume*, pages 74–83, 2013.

[13] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014.

[14] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[15] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *International Symposium on Information Technology: Coding and Computing*

(ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA, pages 586–591, 2005.

[16] J. Blömer and V. Krummel. Analysis of countermeasures against access driven cache attacks on AES. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 96–109. Springer, 2007.

[17] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006*, pages 201–215, 2006.

[18] Andi Drebes, Karine Heydemann, Antoniu Pop, Albert Cohen, and Nathalie Drach. Automatic detection of performance anomalies in task-parallel programs. *CoRR*, abs/1405.2916, 2014.

[19] Faban.org. Faban - helping measure performance, 2014.

[20] Incubator.apache.org. Olio - apache incubator, 2014.

[21] Alexandre Kandalintsev, Renato Lo Cigno, Dzmitry Kliazovich, and Pascal Bouvry. Profiling cloud applications with hardware performance counters. In *The International Conference on Information Networking 2014, ICOIN 2014, Phuket, Thailand, February 10-12, 2014*, pages 52–57, 2014.

[22] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *Proceedings of Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998*, pages 97–110, 1998.

[23] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996*, pages 104–113, 1996.

[24] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[25] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are AES x86 cache timing attacks still feasible? In *Proceedings of the 2012 ACM Workshop on Cloud computing security, CCSW 2012, Raleigh, NC, USA, October 19, 2012.*, pages 19–24, 2012.

[26] M. Neve. *Cache-based Vulnerabilities and SPAM analysis.* PhD thesis, Universite catholique de Louvain, 2006.

[27] M. Neve, J.-P. Seifert, and Z. Wang. A refined look at bernstein's aes side-channel analysis. In F.-C. Lin, D.-T. Lee, B.-S. P. Lin, S. Shieh, and S. Jajodia, editors, *ASIACCS*, page 369. ACM, 2006.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006*, pages 1–20, 2006.

[29] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

[30] Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, and Huseyin Ulusoy. Preventing cryptographic key leakage in cloud virtual machines. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 703–718, 2014.

[31] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[32] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 77–84, New York, NY, USA, 2009. ACM.

[33] C. Rebeiro, M. Mondal, and D. Mukhopadhyay. Pinpointing cache timing attacks on AES. In *VLSI Design*, pages 306–311. IEEE, 2010.

[34] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009.

[35] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011), Hong Kong, China, June 27-30, 2011.*, pages 194–199, 2011.

[36] Sourceware.org. Systemtap, 2014.

[37] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An analytical model for time-driven cache attacks. In A. Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 399–413. Springer, 2007.

[38] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[39] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003*, pages 62–76, 2003.

[40] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*, pages 59–67, 2008.

[41] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M. Swift. Scheduler-based defenses against cross-vm side-channels. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 687–702, 2014.

[42] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 141–150, 2008.

[43] Yuval Yarom and Naomi Benger. Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[44] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732, 2014.

[45] Cemal Yilmaz and Adam A. Porter. Combining hardware and software instrumentation to classify program executions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 67–76, 2010.

[46] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 990–1003. ACM, 2014.

A Experiment Results using All Processes

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure	Timing Overhead
8 Set	Yes	1	Supervised	97.82%	97.59%	97.7%	0.1%
		2		99.60%	99.58%	99.59%	0%
		3		99.91%	99.25%	99.38%	0.5%
		4		99.37%	98.76%	99.06%	0.7%
		5		99.54%	96.49%	97.99%	1%
		6		99.69%	98.02%	98.85%	0.1%
		7		99.77%	91.29%	94.58%	0.01%
	No	N/A	Supervised	98.58%	97.96%	98.25%	0%
	Yes	1	Unsupervised	84.67%	99.66%	91.56%	1.6%
		2		87.19%	99.65%	93%	0%
		3		88.90%	98.89%	93.62%	0.5%
		4		91.67%	99.15%	95.26%	0.8%
		5		83.94%	99.85%	91.39%	1.5%
		6		77.18%	99.58%	87.31%	0.2%
		7		76.66%	99.45%	87.03%	0.1%
	No	N/A	Unsupervised	93.72%	96.67%	95.17%	0.01%

Table 3: Detailed Results of 8 Set Cache Flush using All Processes

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure	Timing Overhead
16 Set	Yes	1	Supervised	99.63%	99.58%	99.60%	0.1%
		2		99.58%	99.36%	99.47%	0.04%
		3		99.58%	99.33%	99.46%	0.08%
		4		99.53%	99.72%	98.11%	0%
		5		99.82%	97.76%	98.78%	0.09%
		6		99.84%	87.99%	93.54%	0%
		7		99.89%	88.21%	93.68%	0
	No	N/A	Supervised	98.81%	98.30%	98.56%	0%
	Yes	1	Unsupervised	83.76%	98.99%	90.74%	0%
		2		87.99%	99.13%	93.07%	1.5%
		3		87.13%	99.80%	93.19%	0.4%
		4		91.95%	99.62%	95.54%	0.9%
		5		82.46%	99.93%	90.20%	0%
		6		79.83%	98.93%	88.74%	1.0%
		7		75.86%	99.66%	86.56%	0
	No	N/A	Unsupervised	93.43%	96.60%	94.99%	0%

Table 4: Detailed Results of 16 Set Cache Flush using All Processes

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure	Timing Overhead
32 Set	Yes	1	Supervised	99.55%	99.51%	99.53%	0.3%
		2		99.42%	99.06%	99.24%	1%
		3		99.42%	99.06%	99.24%	0%
		4		99.59%	96.14%	97.83%	0%
		5		99.80%	96.78%	98.27%	0.6%
		6		99.79%	85.83%	92.28%	0%
		7		99.89%	85.38%	92.07%	0%
	No	N/A	Supervised	98.64%	98.01%	98.32%	0.4%
	Yes	1	Unsupervised	87.96%	99.72%	93.47%	0.4%
		2		87.93%	99.58%	93.39%	1.1%
		3		86.96%	99.18%	92.67%	0%
		4		90.96%	99.83%	95.19%	0%
		5		82.81%	99.85%	90.53%	0.6%
		6		76.90%	99.41%	86.72%	0%
		7		72.33%	99.74%	83.85%	0%
	No	N/A	Unsupervised	93.91%	95.98%	94.93%	0.09%

Table 5: Detailed Results of 32 Set Cache Flush using All Processes

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure	Timing Overhead
64 Set	Yes	1	Supervised	99.71%	99.68%	99.7%	0.4%
		2		99.59%	99.41%	99.5%	1.1%
		3		99.63%	99.38%	99.5%	0%
		4		99.23%	98.91%	99.07%	0%
		5		99.72%	99.00%	99.36%	0.6%
		6		99.25%	93.02%	96.03%	0%
		7		99.82%	91.56%	95.51%	0%
	No	N/A	Supervised	98.82%	98.24%	98.53%	0.1%
	Yes	1	Unsupervised	87.39%	99.68%	93.13%	0.3%
		2		84.92%	99.83%	91.77%	0%
		3		87.48%	99.86%	93.26%	0%
		4		90.91%	99.60%	95.05%	0%
		5		81.33%	99.93%	89.68%	0.2%
		6		77.28%	99.19%	86.87%	0.1%
		7		72.24%	99.80%	83.81%	0.8%
	No	N/A	Unsupervised	92.81%	97.55%	95.12%	0%

Table 6: Detailed Results of 64 Set Cache Flush using All Processes

B Experiment Results using only Cryptographic Process

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure
8 Set	Yes	1	Supervised	82.97%	80.53%	81.73%
		2		81.86%	79.77%	80.80%
		3		83.93%	80.11%	81.98%
		4		24.19%	25.94%	25.03%
		5		22.47%	22.29%	22.38%
		6		18.68%	24.05%	21.03%
		7		16.46%	23.45%	19.34%
	No	N/A	Supervised	86.45%	83.89%	85.15%
	Yes	1	Unsupervised	59.26%	22.52%	32.64%
		2		57.04%	22.05%	31.80%
		3		58.44%	24.12%	34.15%
		4		56.44%	27.95%	37.39%
		5		57.65%	29.04%	38.62%
		6		55.19%	30.77%	39.51%
		7		54.75%	29.90%	38.68%
	No	N/A	Unsupervised	63.82%	26.71%	37.66%

Table 7: Detailed Results of 8 Set Cache Flush using Crypto Process Only

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure
16 Set	Yes	1	Supervised	82.71%	79.96%	81.31%
		2		80.75%	78.39%	79.55%
		3		82.26%	78.85%	80.71%
		4		8.35%	16.01%	10.98%
		5		9.44%	16.88%	12.11%
		6		7.09%	18.33%	10.23%
		7		6.46%	19.65%	9.72%
	No	N/A	Supervised	85.55%	83.01%	84.26%
	Yes	1	Unsupervised	58.17%	21.99%	31.92%
		2		55.64	21.45%	30.97%
		3		56.82%	22.23%	31.95%
		4		57.03%	29.43%	38.82%
		5		59.73%	30.88%	40.72%
		6		56.89%	32.48%	41.35%
		7		54.82%	30.99%	39.60%
	No	N/A	Unsupervised	63.82%	26.71%	37.62%

Table 8: Detailed Results of 16 Set Cache Flush using Crypto Process Only

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure
32 Set	Yes	1	Supervised	82.78%	79.23%	80.97%
		2		81.76%	78.79%	80.2%
		3		83.23%	79.05%	81.09%
		4		56.27%	36.52%	44.3%
		5		62.23%	38.66%	47.69%
		6		70.25%	41.58%	52.24%
		7		66.51%	40.18%	50.09%
	No	N/A	Supervised	85.70%	82.55%	84.10%
	Yes	1	Unsupervised	57.17%	21.18%	30.91%
		2		55.80%	22.57%	32.14%
		3		56.07%	22.76%	32.38%
		4		56.26%	30.46%	39.52%
		5		59.15%	32.40%	41.87%
		6		56.15%	32.60%	41.25%
		7		54.06%	31.05%	39.44%
	No	N/A	Unsupervised	63.24%	27.59%	38.42%

Table 9: Detailed Results of 32 Set Cache Flush using Crypto Process Only

Number of Flushed Cache Set	Workload Classification	Workload Level	Spy Detection	Precision	Recall	F-Measure
64 Set	Yes	1	Supervised	82.35%	78.60%	80.43%
		2		80.43%	77.97%	79.18%
		3		82%	77.31%	79.59%
		4		85.04%	46.81%	60.38%
		5		84.55%	46.58%	60.06%
		6		87.20%	47.31%	61.34%
		7		86.15%	46.55%	60.44%
	No	N/A	Supervised	85.76%	84.32%	84.02%
	Yes	1	Unsupervised	56.17%	21.43%	31.02%
		2		55.41%	21.43%	32.37%
		3		54.12%	21.39%	30.66%
		4		57.80%	30.91%	40.28%
		5		57.97%	31.88%	41.14%
		6		55.33%	32.57%	41.0%
		7		55.39%	31.18%	39.9%
	No	N/A	Unsupervised	62.59%	29.67%	37.70%

Table 10: Detailed Results of 64 Set Cache Flush using Crypto Process Only