# Cache-Based Application Detection in the Cloud Using Machine Learning

Berk Gulmezoglu
Worcester Polytechnic Institute
bgulmezoglu@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

## ABSTRACT

Cross-VM attacks have emerged as a major threat on commercial clouds. These attacks commonly exploit hardware level leakages on shared physical servers. A co-located machine can readily feel the presence of a co-located instance with a heavy computational load through performance degradation due to contention on shared resources. Shared cache architectures such as the last level cache (LLC) have become a popular leakage source to mount cross-VM attack. By exploiting LLC leakages, researchers have already shown that it is possible to recover fine grain information such as cryptographic keys from popular software libraries. This makes it essential to verify implementations that handle sensitive data across the many versions and numerous target platforms, a task too complicated, error prone and costly to be handled by human beings.

Here we propose a machine learning based technique to classify applications according to their cache access profiles. We show that with minimal and simple manual processing steps feature vectors can be used to train models using support vector machines to classify the applications with a high degree of success. The profiling and training steps are completely automated and do not require any inspection or study of the code to be classified. In native execution, we achieve a successful classification rate as high as 98% (L1 cache) and 78% (LLC) over 40 benchmark applications in the Phoronix suite with mild training. In the cross-VM setting on the noisy Amazon EC2 the success rate drops to 60% for a suite of 25 applications. With this initial study we demonstrate that it is possible to train meaningful models to successfully predict applications running in co-located instances.

**Keywords:** Cross-VM Attacks, Machine Learning, SVM, Prime&Probe.

## 1. MOTIVATION

In the last decade the cloud infrastructure has matured to the point where companies, government agencies, hospitals and schools alike have outsourced their infrastructure to cloud service providers. The main benefit of moving to the cloud is the reduction of money spent on IT by pooling servers and storages in bigger cloud services. In many cases, rented servers are instances shared through virtualization among many users. Sharing is the basis for the reduction in the IT costs. Despite the clear cost benefit, given the vast amount of personal and sensitive information kept on shared resources, rightfully security concerns have been steadily growing among cloud customers.

Naturally, the cloud infrastructure has come under the scrutiny of security researchers. The first breakthrough result was reported by Ristenpart et al. [30] who showed that it is possible to co-locate in a controlled manner with possible target instances on commercial public clouds, e.g. Amazon EC2. This work opened the door to a series of investigations that examined the threats from an attacker, i.e. a legitimate cloud user, exploiting cross VM leakage to steal sensitive information. A number of methods have been proposed to steal private keys or valuable information between VMs in IaaS and PaaS clouds [21, 18, 15, 37, 38]. In these works, the cryptographic keys and other sensitive information are stolen by attacker by exploiting leakages at the microarchitectural level, i.e. through the shared cache architecture. Especially, the shared last-level cache (LLC) is a dangerous information leakage source in public clouds. IaaS instance allocation policy commonly allocate an instance per core. This means that the LLC is a shared resource among multiple user instances. Thus by timing his own LLC access times, a user can gleam information on another user's co-located instance's cache access behavior. LLC attacks proliferated to the point that the most recent LLC Prime&Probe attacks do not depend on the de-duplication feature [19, 24] to be enabled to mount cross core cache attacks in public commercial clouds.

Cross-VM leakage attacks are extremely destructive in nature. They require almost no privileges. Anyone can rent an instance on EC2 for a small fee and run an attack code on an instance co-located typically with multiple target instances out of potentially millions of targets. The attack code does only legitimate accesses, e.g. collection of cache access times, for accesses in its own memory/application space. Thus, Cross-VM attacks pose a great threat. Potentially, one could automate the attack and mine the entire compute cloud for cryptographic keys. There are practical difficulties in carrying out such attacks on a mass scale. Cross-VM security attacks on public clouds require a sophisticated methodology to extract the sensitive information from data. For instance,

*ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates*
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3052973.3053036

the cache pattern is extracted and by using personal effort the relation between pattern and key is established [18]. This makes discovery of vulnerabilities, a manual process, rather costly and time-consuming. Cryptographic library designers experience a similar difficulty. Cryptographic libraries are constantly patched for newly discovered leakages and emerging vulnerabilities. This in itself is a painstaking process requiring careful inspection of the code for any potential leakage for a target platform[1]. Software bugs may result in secondary leakages confounding the problem further. Thus, in practice, even constant execution flow/time implementation may be compromised due to bugs.

With the growing complexity of cryptographic libraries, or more broadly of code that handles sensitive data, it becomes impossible to manually verify the code for leakages across the numerous platforms exhaustively. Clearly, there is a great need for automated verification and testing of sensitive code against leakages. Firstly, Brumley et al. [10] proposed vector quantization and HMM to classify ECC ops with respect to L1-D cache. Then, an automated profiling attack on LLC was introduced by Gruss et al. [13]. In this work, the access pattern of different events are first extracted in a non-virtualized (less noisy) environment. The attacker learns the cache access templates from the cache. During an attack the new data is compared against the learned templates. While this is a worthy effort, machine learning (ML) algorithms have advanced to the point where they offer sophisticated solutions to complicated recognition, classification, clustering, and regression problems. For instance, image and speech recognition, sense extraction in text and speech [28], recommendation systems and search engines, as well as malicious behavior detection [11]. Further, cryptographers recently started to consider machine learning algorithms for side channel analysis, [23, 16].

In this work we take another step in this direction. We are motivated by the need for automation in cross-VM leakage analysis. Our goal is to minimize the need for human involvement in formulating an attack. While more sophisticated techniques such as deep neural networks can solve more complicated problems, they require significantly more training data and take longer. Instead here we focus on more traditional ML techniques for classification. In particular, we are interested in automating classification of applications through their cache leakage profiles in the Cross-VM setting. A successful classification technique would not only compromise the privacy of a co-located user, but could also serve as the initial discovery phase for a more advanced follow-up high precision attack to extract sensitive information. To this end, in this work we first profile the cache fingerprints of representative benchmark applications, we then identify the minimal processing steps required to extract robust features. We train these features using support vector machines and report success rates across the studied benchmarks for experiments repeated for L1 and LLC. Finally, we take the attack to AWS EC2 to solve a specific problem, i.e. we use the classification technique to show that it is possible to detect other co-located VMs. We achieve this by sending ping requests to open ports by simultaneously monitoring LLC on Amazon EC2. If the ping receiver code is detected running on the co-located instance we infer co-location with the targeted IP.

## Our Contribution

We present a study in automation of cache attacks in modern processors using machine learning algorithms. In order to extract fine grain information from cache access patterns, we apply frequency transformation on data to extracted fingerprints to obtain features. To classify a suite of representative applications we train a model using a support vector machine. This eliminates the need for manually identifying patterns and crafting processing steps in the formulation of the cache attack. In our experimental work, we classify the applications bundled in the Phoronix Test Suite. Note that we do not have any information about the content of the code and nor have we studied any internal execution patterns.
In summary, this work

- for the first time implements machine learning algorithm, i.e. SVM, to profile the activity of other users on the cloud
- extracts the feature vectors from cache access data for different types of applications using a straightforward FFT computation,
- demonstrates that there is no need for synchronization between spy and target to profile an application while SVM based approach is implemented,
- shows that targeted co-location is achievable by sending ping requests on Amazon EC2 if the targeted IP is known by spy

The rest of the study is divided as follows. We first review the related work and give the background knowledge in Section 2. The approach is presented in Section 3. The experiment setup and results are explained in Section 4 and the paper is concluded in Section 6

## 2. BACKGROUND

In this section we give a brief overview of the related work in terms of cache attacks and several implementations of machine learning techniques in different side channel analysis.

### 2.1 Related Work

**Co-location detection techniques:** In 2009, Ristenpart et al. [30] demonstrated the possibility of the co-location between attacker and victim in public IaaS clouds. After two years, Zhang et al. [36] detected the co-location by simply monitoring the L2 cache if attacker and victim reside on the same core. In 2012, Bates et al. [6] showed that if the network traffic is analyzed it is possible to detect the co-location. In 2014, it is shown that deduplication enables the co-location detection in Paas clouds by Zhang et al. [38]. Recently, Varadarajan et al. [33] and Inci et al. [17] showed that the co-location detection is still possible on Amazon EC2, Google Compute Engine and Microsoft Azure using memory bus locking.

**Cache Attacks:** Cache attacks are widely used to extract information from cryptographic libraries. In 2003, Tsunoo et al. [32] presented a cache attack on DES using cryptanalysis. In 2004, AES cache attacks were firstly presented by Bernstein [7] using microarchitectual timing differences for different look-up table positions in cache. In the same year, Osvik et al. [27] implemented two new cache attacks

[1] A code that is considered secure on one platform, may not be on another due to microarchitectural differences.

(Evict+Reload and Prime&Probe) on AES. Both attacks recovered the AES encryption key with different number of encryption. After it is shown that it is possible to recover AES encryption key the community focused on analyzing the potential thread of cache attacks on both AES and RSA.

In 2006, Bonneau et al. [8] implemented cache collision attacks in the last round of AES. In 2007, the similar collision attacks are exploited by Aciicmez et al. [3]. In the same year, Aciicmez et al. [4] the first attack against RSA was implemented by monitoring instruction cache accesses. In 2011, Gullasch et al. [14] presented a new cache attack on AES namely, Flush and Reload.

With the increasing popularity of cloud computing systems, the attacks are implemented on public clouds. In 2012, Zhang et al. [37] presented the first cache attack on cloud by recovering an ElGamal encryption key in the same core.

In 2013, the first cross-core cache attacks are studied. Yarom et al. [34] used the same technique in [14] to recover a full RSA key in LLC. In 2014, Irazoqui et al. [22] recovered first AES key among cross-VM scenario using Flush and Reload. The Flush and Reload attack is also implemented in different scenarios such as on PaaS clouds and cache template attacks [38, 13].

However, the Flush and Reload attack is applicable if deduplication is enabled among VMs. It is known that deduplication is disabled on public clouds. In order to overcome this difficulty, Liu et al. [24] and Irazoqui et al. [19] implemented a new Prime&Probe attack in the LLC by using hugepages. Recently, Inci et al. [18] showed the applicability of this attack by stealing 2048 bit RSA key on Amazon EC2 cloud. At the same time, Oren et al. [26] implemented Prime&Probe attack in javascript to monitor different web browser scenarios.

**Exploiting cache slice selection methods:** In Intel processors there are two types of slice selection methods. The first one is the linear slice selection algorithm where the same lines can be used to create eviction sets by simply changing the set number. The recovering techniques for linear cache selection was presented in Irazoqui et al. [20] and Maurice et al. [25] using the coincidence of the functions across processors. Recently, Yarom et al. [35] recovered a 6 core slice selection algorithm using the time differences of cache in different cores. Finally, 10-core Intel architecture is reverse engineered in [18] by creating many lines and analyzing the possible algorithms.

**Machine Learning Techniques on side channel analysis:** Firstly, machine learning techniques were applied to side channel analysis in 2011 by Lerman et al. [23]. In this work, the relation between 3DES encryption and power consumption was studied using dimensionality reduction and model selection. For dimensionality reduction classical Principal Component Analysis (PCA) was implemented and for model selection Self Organizing Map (SOM), Support Vector Machine (SVM) and Random Forest (RF) techniques were compared. In the same year, Hospodar et al. [16] applied Least Square Support Vector Machine (LS-SVM) to extract information from power consumption of AES. In 2012, Zhang et al. [37] implements SVM to classify multiplication, modular reduction and square operations to extract the ElGamal decryption key. In this work, Hidden Markov Model (HMM) is applied to probably estimates of SVM to reduce the noise and the success rate becomes higher.

## 2.2 Prime&Probe Technique

In the modern computer architecture, it is not possible for users to see the physical address of a line because of the security issues. Therefore, the virtual address is translated from the physical address and it is visible to users. In virtual address the first 12 bits are exactly same with the first 12 bits of physical address. However, this is not enough to find the corresponding cache set for the line in LLC. Thus, it is not possible to create an eviction set with regular 4KB pages in LLC.

The Prime&Probe technique is the most widely applicable profiling technique on the cloud since all major Cloud Service Providers (CSPs) have disabled deduplication, making Flush and Reload attacks infeasible. To achieve an eviction set in LLC the spy needs to know more than 12 bits of the physical line. If Huge pages (2MB) are allocated by the spy, it is possible to know the first 21 bits of the line which is enough to know the corresponding set in LLC. After finding the eviction set for the desired set, the eviction can be implemented. The Prime&Probe profiling is divided into three main stages:

1. **Prime stage**: This stage is used to create an eviction set. To create an eviction set the spy generates distinct lines which reside on the monitored set. The number of lines in the eviction set is equal to number of ways in the monitored set. After all lines accessed by the spy the eviction set is ready.

2. **Waiting stage**: In this stage, the spy waits for the target to evict some lines from the primed set. The waiting time is crucial to determine the resolution of the profiling. While the time is increasing the frequency and resolution are getting lower.

3. **Probe stage**: In the probe stage, the spy accesses the addresses used in the prime stage. If the monitored set was not accessed by another process, no data has been evicted; all accesses result in a cache hit, giving a low access time. If another process has accessed the monitored set, its data must have evicted at least one of the lines of the spy's data. Hence, the probe access will include accesses to memory, resulting in measurably higher access times.

In native and cloud environment experiments we used non-linear slice selection algorithm since EC2 Cloud uses 10 core non-linear slice selection algorithm. In non-linear slice selection algorithm for each set the eviction set should be created by implementing the algorithm. This makes the process harder because to find the eviction set for all sets in LLC by hand takes huge amount of time. Therefore, the algorithm in [15] is implemented in 10 core machine to create LLC eviction sets faster. The ratio of noisy sets should remain the same for linear and non-linear slice selection algorithms. Hence, we believe the proposed work is applicable to all Intel Ivy bridge processors.

## 2.3 Support Vector Machine (SVM)

SVM is a data classification technique used in many areas such as speech recognition, image recognition and so on [28, 5]. The aim of SVM is to produce a model based on the training data and give classification results for testing data.

Firstly, SVM is built for binary classification. For a training set of instance-label pairs $(x_i, y_i)$, $i = 1, \ldots, k$ where

$x_i \in R^n$ and $y \in \{1, -1\}^k$, SVM require the solution of the following optimization problem:

$$\begin{aligned} &\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{k} \xi_i \\ &\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i \\ &\qquad \text{where } \xi_i \geq 0 \end{aligned} \tag{1}$$

The function $\xi$ maps the training vectors into a higher dimensional space. In this higher dimensional space a linear separating hyperplane is found by SVM where $C > 0$ is the penalty parameter of the error term. For the kernel function in our paper we use linear kernel:

$$K(x_i, x_j) = x_i^T x_j \tag{2}$$

In our work, we have many application to classify. Therefore, multi-class SVM library LIBSVM [12] is used.

In addition, we apply the cross-validation (CV) technique. In K-fold CV, the training data $D$ is partitioned into N subsets $D_1, \dots, D_N$. Each data in $D$ is randomly separated to each subset with equal size. So, we define $D_{t_i} = \bigcup_{j=1,\dots,N(j \neq i)}$ where $D_{t_i}$ is the union of all data except those in $D_i$. For each subset a model is built by applying the algorithm to the training $D_{t_i}$. The average of N results are evaluated as cross-validation (test) performance.

# 3. METHODOLOGY

In this section, we show how machine learning can be used on cache profiles to detect running programs. One specific use case is the detection of the ping service, which can serve as an implicit co-location test.

## 3.1 Extracting Feature Vectors from Applications on Cache

Our thesis is to show that programs have unique fingerprints in cache and it is possible to learn and classify application fingerprints using ML algorithms with a high accuracy. The proposed approach starts by creating profiles for every software using the Prime&Probe technique. This way, dynamic and static functions of the application are detected, resulting in fairly reliable fingerprints. The raw cache timing traces are first turned into hits and misses, followed by a Fourier transform. Performing a Fourier transform on the cache profiles removes the need for tight synchronization and makes the approach more resilient to noise. The FFT output can then directly be fed into a machine learning method of choice. The process to obtain application fingerprints is visualized in Figure 1. Our approach differs from previous works in cache-based information extraction in several ways:

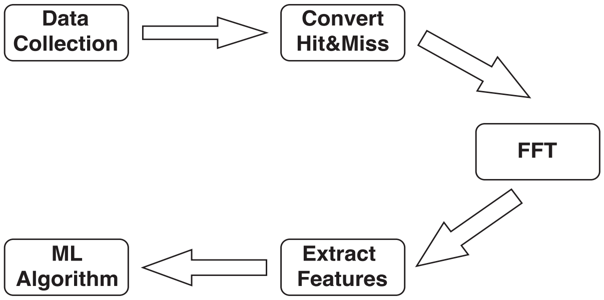

**Figure 1: The flow chart of the approach for both L1 and LLC profiling**

**Fourier Transform** Most previous works [13, 18, 22], assume the monitoring process to be synchronized. The synchronization is handled by triggering the event, then the profiling phase is started. However, it is not trivial when the monitoring process and the target do not have communication. In addition, the functions periodically accessed in the application would give a certain information which could be exploited by using Fourier transform. Therefore, we transform the data to frequency domain from time domain in order to eliminate a strong assumption like synchronization and to extract the periodic functions' cache accesses as fingerprints of the applications.

**No Deduplication** Deduplication enables incredibly powerful side channel attacks [13, 9, 29], most prominently the Flush and Reload technique [34, 13]. However, public cloud service providers are aware of this issue and have disabled deduplication. Therefore, it is impossible to track data of other VMs in shared memory in IaaS and most PaaS clouds. Hence, the Prime&Probe technique is preferred to implement in our scenario instead of the Flush and Reload method to eliminate the strong assumption for deduplication. Prime&Probe technique is simply based on fill all ways in the monitored LLC set by enabling Huge pages which is possible in all public clouds.

The resolution of the resulting analysis is lower than Flush and Reload method in LLC, however the results show that after training enough data it is efficient to detect programs used by other co-located VMs.

**Detecting Dynamic Code** Our method does not make any assumption on whether code is dynamic, static or a shared function. Instead, we profile one of the *columns* in the cache, as shown in Figure 2. This means the location of a line in LLC might change from one run to another run if the function is dynamic. However, the offset bits ($o$) never change therefore, it resides on one of the set-slice pairs solving $s \bmod 64 = o$.

**Long profiles** Our method shows that even if the entire process of a program is not profiled, the spectral density of a small part of the program can give enough information to detect the program (in fact, the length of the analyzed programs varies from two seconds to 3.5 hours).

Our approach starts by creating cache profiles for every application by using the Prime&Probe technique in the same core to monitor all L1 cache sets. The analysis of L1 cache leakage provides a very high-resolution channel, thereby describing a best-case scenario for the learning technique. In addition, the L1 cache experiments provide valuable information which cache sets are actively used by the application. This information can be used as a preparatory step for LLC profiling to find the corresponding active sets in LLC. After the data is collected for a set of application, the Fourier transform is applied to extract the feature vectors subsequently used in an ML algorithm.

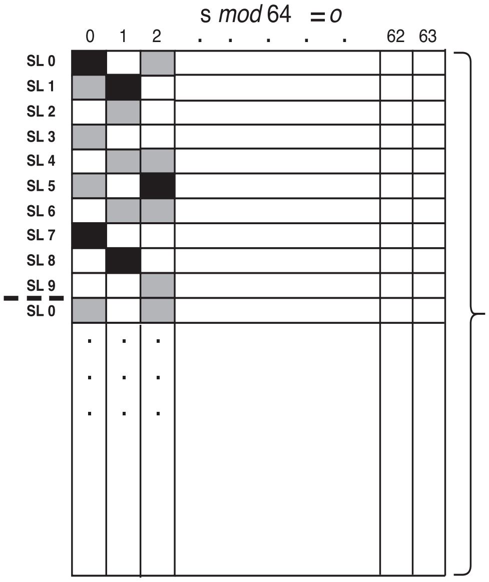

**Figure 2: Visualization of 10 core LLC. Gray set-slice pairs are noisy, white set-slice pairs are unused sets and black set-slice pairs are actively used by target application.**

**Table 1: Symbol Descriptions**

| Symbols | Description |
|---|---|
| $S_{L1}$ | Number of sets in L1 cache |
| $N_T$ | Number of traces collected |
| $N_D$ | Number of data sets per test |
| $N_S$ | Number of applications |
| $N_C$ | Number of cores |
| $N_A$ | Number of active sets |
| $\tau_o$ | Outlier threshold for samples |
| $\tau_c$ | Hit&Miss threshold |
| $F_s$ | Sampling Frequency |
| $F_{CPU}$ | CPU frequency |
| $T_{cc}$ | Prime&Probe time |
| $L_f$ | Length of fingerprint |

## 3.2 Extracting feature vectors from L1 cache

We assume that the number of L1-instruction and L1-data cache sets are $S_{L1}$ for either cache. In L1 cache, the data and instruction cache are monitored separately, while profiling process and target application are running in the same core. The overall process to monitor L1 sets and creating feature vectors for different applications is given in Algorithm 1.

For L1-data and L1-instruction monitoring, a total of $N_T$ traces is collected per set for each data set. Therefore, for each data set we have $S_{L1} \cdot N_T$ traces in total. After collecting several data sets, the total number of traces is equal to $S_{L1} \cdot N_T \cdot N_D$ where $N_D$ is the total number of data sets per application collected by the spy.

The outliers in the data should be filtered before the raw data ($R$) is converted to binary data ($B$). Hence, the L1-data and L1-instruction sets are monitored in idle case and base Probe values are recorded. The outlier threshold ($\tau_o$) and binary conversion threshold ($\tau_c$) are obtained based on the idle values.

The Probe timings are compared to $\tau_o$ and $\tau_c$. The values are higher than $\tau_o$ are set to median value of idle case of that set to get rid of the outliers. The conversion from $R$ to $B$ is also implemented by comparing with $\tau_c$. If the Probe time is higher than $\tau_c$, then the trace is converted to 1, implying an access to the cache set. If it is below than $\tau_c$, the trace is converted to 0, implying no cache access. The resulting binary trace is converted by using Fourier transform.

In the transformation phase, the sampling frequency should be computed. In order to calculate the sampling frequency ($F_s$), the total Prime&Probe time for monitored set is computed in clock cycle ($T_{cc}$), then the CPU frequency ($F_{cpu}$) is divided by $T_{cc}$ to get $F_s$ for L1-data and L1-instruction cache. We assume that the sampling frequency is same for all sets in L1-data and L1-instruction cache. To calculate the frequency components of binary data, $F_s$ is used in FFT and the length of the outcome is $N_T$. However, the result has two symmetric sides of which only the first half is used as a fingerprint. Therefore, the length of a fingerprint obtained from one data set is $N_T/2$.

**Algorithm 1** L1 Profiling Algorithm

```
$F_s = F_{CPU}/T_{cc}$
for $i$ from $1$ to $N_S$ do
    for $j$ from $1$ to $N_D$ do
        for $k$ from $0$ to $S_{L1} - 1$ do
            for $l$ from $1$ to $N_T$ do
                if $R(i,j,k,l) \geq \tau_o$ then
                    $R(i,j,k,l) = median(R(i,j,k,1:N_T))$
                end if
                if $R(i,j,k,l) \geq \tau_c$ then
                    $B(i,j,k,l) = 1$
                else
                    $B(i,j,k,l) = 0$
                end if
            end for
            $L(i,j,k) = FFT[B(i,j,k,1:N_T)]$
        end for
        $L_f^{i,j} = L(i,j,0:S_L - 1)$
    end for
end for
```

For each process there are $S_{L1}$ different fingerprints hence, these fingerprints are concatenated from set 0 to set $S_{L1} - 1$ sequentially. Thus, the total length of the fingerprint of a process is $L_f = S_{L1} \cdot N_T/2$. If the total number of data sets is $N_D$ then, the size of a training matrix of a process is $N_D \cdot L_f$.

The SVM then processes all matrices combined and labeled from 1 to $N_S$ where $N_S$ is the number of different applications. The final success rate is computed using 10-fold cross-validation.

## 3.3 Extracting feature vectors from LLC

Next, we apply the approach on LLC leakage. LLC has the advantage that is accessible for all processes on the same system. Hence, as long as the monitored process runs on the same system as the monitor, the side channel is accessible, even if the two processes run in different VMs.

After finding the most used L1 set, the corresponding sets in LLC should satisfy $s$ mod $64 = o$ where $o$ is the L1 set

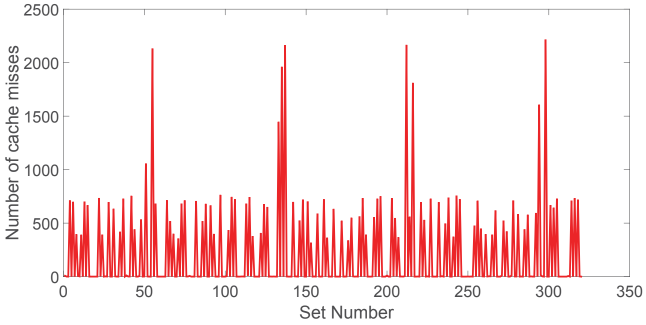

**Figure 3: Eliminated noisy sets in LLC**

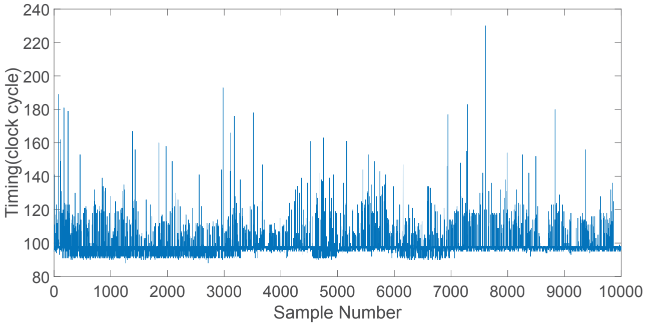

**Figure 4: One of the active sets for an application**

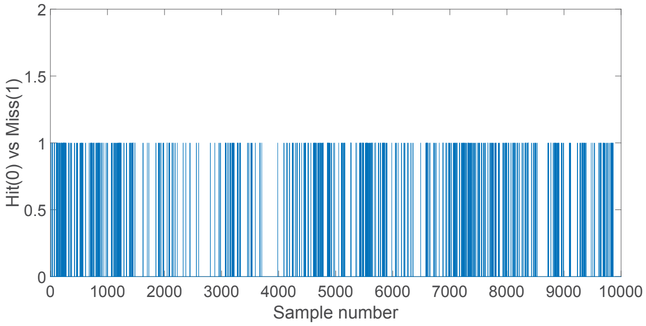

**Figure 5: Hit(0) and miss(1) graph of an active set**

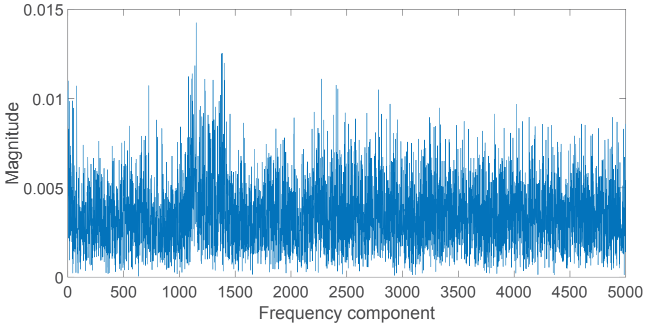

**Figure 6: Frequency components of an active set**

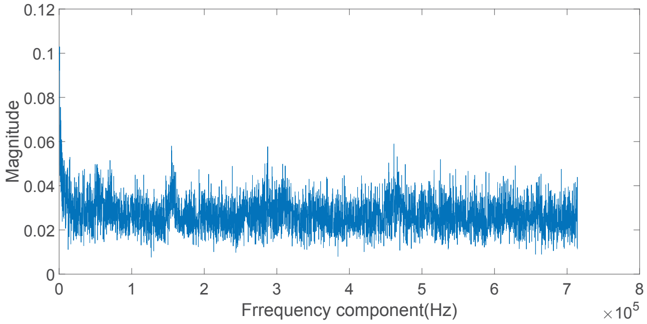

**Figure 7: Combination of frequency components of all active sets**

number. The number of corresponding sets vary with the number of cores $N_C$. In total, the number of LLC set-slice pairs on current Intel CPUs can be determined by

$$S_{L3} = 2^{N_{LLCB}-N_o} \cdot N_C \tag{3}$$

where $N_{LLCB}$ is the number of LLC bits and $N_o$ is the number of offset bits. After the eviction set for each set-slice pair is created by using the algorithm [15], the Prime&Probe profiling starts. For LLC profiling $N_T$ is same with L1 profiling and after $N_T$ traces are monitored in one set-slice pair, the next set-slice pair is profiled. The reason behind this is to increase the temporal resolution for each set-slice pair which is crucial to catch dominant frequency components in frequency domain.

After collecting $N_T \cdot S_{L3}$ data, the same process in L1 profiling is applied to LLC traces to get rid of the outliers. Before the binary data is derived from the raw data, the noisy set-slice pairs need to be eliminated. For this purpose, the number of cache misses are calculated in idle case and if the number of cache misses are higher than 1% of $N_T$ that set-slice pair is marked as noisy, as shown in Figure 3. After noisy sets are determined all of them are excluded from the next steps since the spectral density of these sets is not stable.

The active set-slice pairs are determined by checking the number of cache misses in the data. If the number of cache misses is higher than 3% of $N_T$ in Figure 4, then the set-slice pair is marked as an active set. After all active sets are derived, they can be converted to binary data with the same process in L1 profiling in Figure 5 before the Fourier transform starts.

For the Fourier transform $F_s$ should be calculated for LLC sets. $F_s$ is lower in LLC profiling since the number of ways in the sets are higher than L1 sets and the access time to lines reside on LLC is greater than L1 lines. Therefore, the total Prime&Probe time for each set-slice pair should be calculated and the average of all of them are used as LLC $F_s$. After $F_s$ is calculated, the active sets are transformed to frequency domain in Figure 6. The number of frequency components per $N_T$ is same with the L1 profiling.

The number of active sets ($N_A$) may vary for each process therefore, the concatenated active sets have different length for each software. To solve this issue we propose to combine all frequency components of active sets. All frequency components are summed up element-wise and a fingerprint is obtained from each data set. In LLC profiling the length of the fingerprint is smaller than L1 profiling because in LLC scenario we cannot concatenate all active sets.

After obtaining all data sets for each application the total size of matrix for LLC training data is $N_D \cdot (N_T/2)$. The

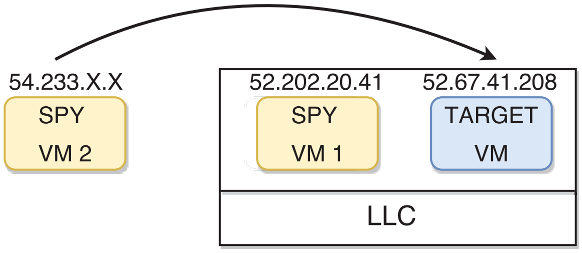

**Figure 8: The scenario for ping detection on Amazon EC2**

SVM algorithm is applied as in the L1 profiling case and the results are recorded.

### 3.4 Targeted co-location by ping detection on the cloud

Another use case of the described methodology is the detection of whether or not a specific application is being executed. For this purpose we propose to detect ping requests sent to a target VM. We then try to detect the execution of the ping response process to verify and detect co-location with that target VM. In order to detect the co-location on the cloud, different types of covert channels such as LLC [17] and memory bus locking [33] have been used. These methods can be effective to verify the co-location between spy and target VMs. Our method also uses LLC, but, due to the omnipresence of ping support, this method is very widely applicable. The scenario is as follows: the spy VM monitors LLC sets by Prime&Probe to check the co-location with the target VM in the same cloud region. Another collaborating process of the spy sends ping requests to the target VMs with a certain frequency. These ping requests trigger executions of the ping service, which is then observable by the spy VM.

The used approach is similar to the previous cases: The monitored sets are determined by $s \bmod 64 = 0$. The reason behind this is the ping receptions are seen random sets. Therefore, we find that it is sufficient to monitor these sets to detect the ping. The steps to detect ping on the cloud are as follows:

1. Spy VM1 finds the noisy sets and excludes them from $S_{L3}$ sets in VM1
2. Ping requests are sent by spy VM2 with a certain frequency
3. Spy VM1 begins to implement Prime&Probe on remaining sets
4. Spy VM1 determines the active sets in LLC
5. Fourier Transform is applied to the active sets
6. The frequency components are compared with the ping frequency

In our method, first the active IPs and the open ports should be found. In Amazon EC2 every region has different IP ranges. We focus on South America region and the IP range is documented [1]. Open ports can be found using the *nmap* tool.

There are two types of ping namely *hping* and *ping* commands. The *hping* command is more useful since specific ports can be pinged such as port 22 which is used for SSH connection. Furthermore, the frequency of ping requests for *hping* command can be set higher than in the *ping* command. There is a need to have more frequent ping requests, as high-frequent calls to ping strengthen the LLC profile and thus decrease the number of traces needed to detect the ping requests in LLC. Therefore, we used *hping* in our Amazon EC2 experiments.

## 4. APPLICATION DETECTION RESULTS

In this section, we explain the experiment setup used to collect data and make our scenario applicable.

### 4.1 Experiment Setup

For the experiments, we have used the following two setups;

– **Native Environment:** In this setup, the applications are running on a native Ubuntu 14.04 version with no virtualization. The processor is a 10 core Intel(R) Xeon(R) E5-2670 v2 CPU clocked at 2.50 GHz. The purpose of this scenario is to run experiments in a controlled environment with minimal noise and to show the high success rate of our methods. In addition, this processor is the same type of processor mainly used in Amazon EC2 cloud.

– **Cloud Environment:** In this setup, Amazon EC2 cloud servers are used to implement our experiments in a cross-VM scenario. In Sao Paulo region, the processors are same with the one used in native environment with a modified Xen hypervisor. The instance type is *medium.m3* which has 1 vCPU. The aim of this setup is to show the huge thread of our scenario in a public cloud. In this setup, there are two co-located VMs in the same physical machine sharing the LLC which is verified by the techniques [17].

To evaluate our approach on a broad range of commonly used yet different applications, we decided to use the Phoronix test benchmarks as sample applications for classification [2]. We performed classification experiments on these applications in three different scenarios. As baseline experiments we first performed the experiments in the above-described native scenario, both by monitoring L1 cache leakages and also by monitoring LLC leakages. The former shows the potential of L1 cache leakages if they are accessible. The latter assumes a realistic observation scenario for any process running on the same system. Finally, we performed the same experiments on Amazon EC2 cloud to show the feasibility in a noisy and cross-VM scenario. In this public cloud scenario, only LLC is profiled to classify benchmarks since each VM has only one thread in the core and they do not reside on the same core. For both L1 cache and LLC experiments, our methodology is applied to 40 different Phoronix benchmark tests in including cryptography, gaming, compressing, SQL, apache and so on Appendix 9. Last but not least, we present a scenario where we only try to detect the presence of a single application, the ping detection described in Section 3.4.

## 4.2 Application Detection in Native Environment

We first performed experiments in the native environment.

### 4.2.1 *Monitoring L1 Cache*

In native case, first we implemented our profiling on L1 cache. There are two types of cache structure namely, data and instruction. Therefore, in our experiments we profiled each of them separately. In our processor there are $S_{L1} = 64$ sets for each L1-data and L1-instruction cache and the sets are 8 way associative.

The profiling and application code run on the same core to detect misses in L1 cache. Hence, the hyper-threading feature of Intel processors is used. Before the training data is collected, an idle case of L1-data and L1-instruction sets are monitored and base Probe values are recorded. For L1-data the base value is around 65 clock cycles and for L1-instruction it is around 75 clock cycles. Hence, the outlier threshold is chosen as $\tau_o = 150$ for both data and instruction cache. For the conversion from raw data to binary data the threshold value is $\tau_{o,d} = 80$ for data cache and $\tau_{o,d} = 90$ for instruction cache. The number of traces collected per set for each data set is $N_T = 10,000$. Therefore, the total number of traces is equal to 640,000 which belongs to one data set for L1-instruction or L1-data.

To compute the sampling frequency, we checked the total Prime&Probe time and it is almost same for all sets in L1 cache which is around $T_{cc} = 200$ clock cycle. Hence, the sampling frequency is $F_s = 2.5GHz/200 = 12.5MHz$ for L1 cache profiling. $F_s$ for L1 cache is higher than LLC profiling because the number of ways in L1 sets is smaller than LLC sets and accessing to L1 cache lines is faster than LLC lines. Thus, the resolution of L1 profiling is higher than LLC profiling which results more distinct feature vectors and high success rates in ML algorithm.

After $F_s$ is determined, FFT can be applied to traces. The outcome of FFT is $N_T/2$ which is equal to 5,000 frequency components in our case. This process is applied to all 64 sets in data and instruction cache for each test. Hence, the feature vector of a test consists of 320,000 frequency components after all sets are concatenated. The number of data sets per test is $N_D = 60$ which means the training data is a matrix of the size $2,400 \times 320,000$.

To classify the training data, first 10-fold cross validation is implemented in SVM. For cross-validation we implement both C-SVC and nu-SVC SVM types in the LIBSVM library. Our results show that C-SVC gives better success rates, so we preferred this option. For the kernel type, the linear option is chosen since the success rate is much higher than for the other options. After these options are chosen kernel parameters and the penalty parameter for error are optimized by LIBSVM. In both training and test phases the chosen parameters are used to implement SVM. Therefore, there is no user interaction to choose the best parameters and the steps are automated.

In cross-validation experiments, we show the effect of number of L1 sets on success rate. If only 1 set is used to generate the training model, the cross-validation success rate is 46.8% for instruction and 60.71% for data cache. With the increasing number of sets, the cross-validation success rate for data and instruction cache is increasing to 95.74% and 97.95%, respectively in Figure 9.

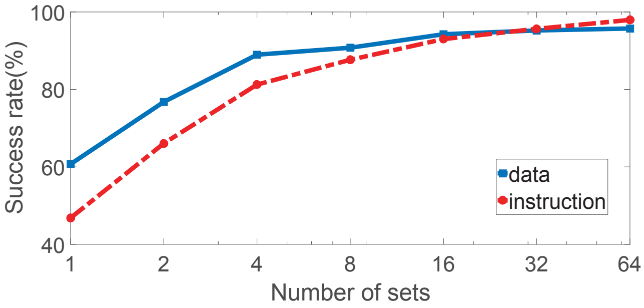

**Figure 9: Success rate graph for varying number of sets to train the data**

For training and individual success rate of test, 60 data sets per test are trained where the SVMMODEL is obtained with C-SVC and linear kernel options. With the cross-validation technique, the success rate for instruction cache is higher than data cache. The reason behind this is some of the Phoronix tests do not use L1-data cache however all tests use L1-instruction cache. Therefore, extracting the feature vectors for tests in instruction cache is more successful than L1-data cache.

The results also show that the cross-validation success rate is 98.65% if all information in L1 cache (both instruction and data) is used in the machine learning algorithm. To achieve this success rate we used all 64 cache sets and in total we have $50 \times 640,000$ size feature vectors per test. Therefore, the size of training data is $2,000 \times 640,000$.

### 4.2.2 *LLC Results*

L1 profiling is not realistic in the real world since the probability of two co-located VMs in the same core is really low. Therefore, before switching to public cloud we implemented our attack in LLC with a cross-core scenario. The number of cores is $N_C = 10$ in our processor and the number of set-slice pairs solving the equation in 3 $s$ mod 64 $= o$ is $S_{L3} = 2^5 \cdot 10 = 320$ where $N_{LLCB} = 11$ because of 2,048 LLC sets in total and the number of offset bits is $N_o = 6$. $o$ is the set number which is the most used one in L1 profiling for that test. Therefore, we have 320 set-slice pairs in total to monitor.

Before collecting data for every test, the idle case of each set is monitored to determine the base value ($\tau_b$). $\tau_b$ changes between 90 and 110 clock cycle among different set-slice pairs. Hence, for each set-slice pair $\tau_b$ is different. The outlier threshold ($\tau_o$) is 250 clock cycle. The threshold value ($\tau_c$) for the conversion from raw data to binary data is $\tau_b$+15 clock cycle. After obtaining the binary data, it is trivial to find the noisy sets. If the number of cache misses is higher than 100 in a set-slice pair, it is marked as noisy. These noisy sets are not processed when the data is collected.

While collecting the training data 10,000 traces are collected per set-slice pair. The active sets are determined by checking the number of cache misses in each set-slice pair excluding the noisy sets. If the number of cache misses is higher than 300, then that set-slice pair is marked as active and they are included in Fourier transform.

The Prime&Probe timings change between 1,800 and 2,200 clock cycle so the sampling frequency ($F_s$) is taken 1.3 MHz. After FFT is applied to active sets, the left symmetric side of the outcome is recorded. The length of the fingerprint

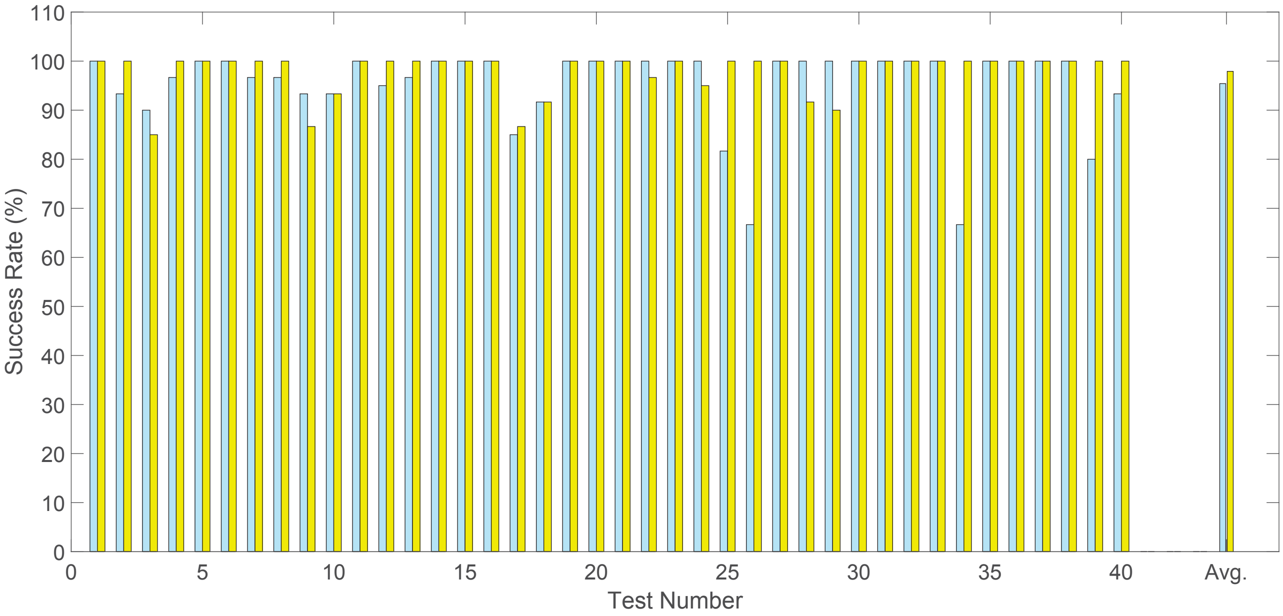

**Figure 10: Success rate for different tests in L1-data (blue) and L1-instruction (yellow). The last bar represents the average of success rates for 40 tests**

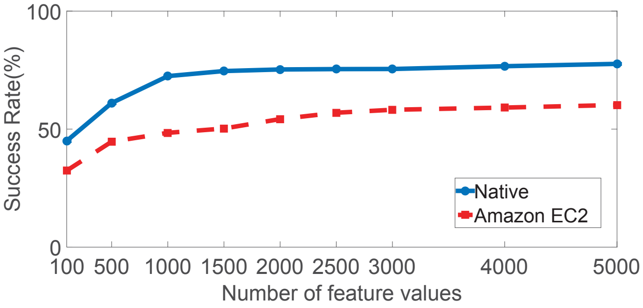

**Figure 11: LLC success rate with varying number of frequency components**

for a set-slice pair consists of 5,000 frequency components. If there are 6 active sets for a test, the fingerprint of each active set are combined by element-wise in each data set and final fingerprint is obtained from one data set per test.

For LLC experiments, we used 40 different benchmark tests to profile in LLC. The number of data set per test is 50 and the length of vector for each feature vector is 5,000. After collecting the data the training model is generated and the cross-validation is applied to training data.

For the cross-validation, same options in L1 profiling are used in SVM. The success rate for LLC test in average is 77.65% with 5,000 frequency components. With the decreasing number of frequency components the success rate drops to 45% in Figure 11.

The details of success rates for different tests are presented in Figure 12 by using 10-fold cross-validation technique. The results are obtained from 5,000 frequency components and 60 data sets per test. The lowest recognition rate is 13% for *GMPBENCH* test since the success rate for this test is low in L1-data cache in Figure 10.

## 4.3 Application Detection on EC2 Cloud

To show the applicability of ML technique to real world, we also perform our profiling method on Amazon EC2. The challenges of performing the experiments on a public cloud are hypervisor noise and the noise of other VMs in the monitored sets. Therefore, some set-slice pairs are marked as active even if those pairs are not used by target VM. Redundant cache misses in the active sets also pose a problem. During Fourier Transform, these cache misses may cause shifts in frequency domain. To overcome these difficulties, SVM technique is applied to the data, and as a result, the success rate gets higher.

The number of tests decreases in cross-VM scenario since some tests do not work properly and some of them have installation problems on Amazon EC2. Thus, the number of tests used in this experiment decreases to 25. To classify the different benchmark tests, same process in LLC profiling is used, then training data is processed in SVM. The result is lower than native case because of the aforementioned types of noise. The 10-fold cross-validation result is 60.22% in Figure 11 with 5,000 frequency components. This result shows that on public cloud the classification success rate drops with increasing noise.

The success rates for individual tests change between 16% and 100% in Figure 13. The success rate decreases when the hypervisor and other VMs noise affect the cache miss patterns. Even though the success rate is lower than native scenario, this result demonstrates the applicability of out method in the cloud platform.

## 4.4 Ping detection on EC2

To detect the co-located VMs with spy VM, ping requests are sent by one of the VMs controlled by the spy in the same region. The purpose of this is to decrease RTT and increase the frequency of ping requests. At the same time, spy VM 2

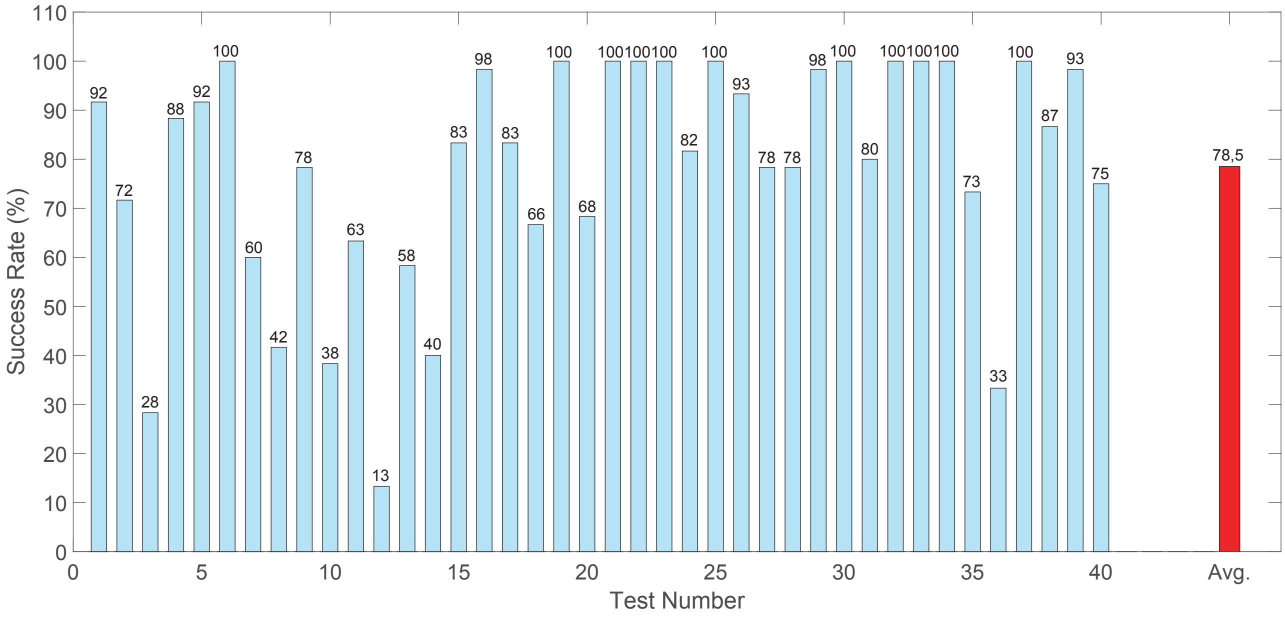

**Figure 12: LLC success rates for different tests in native scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all tests**

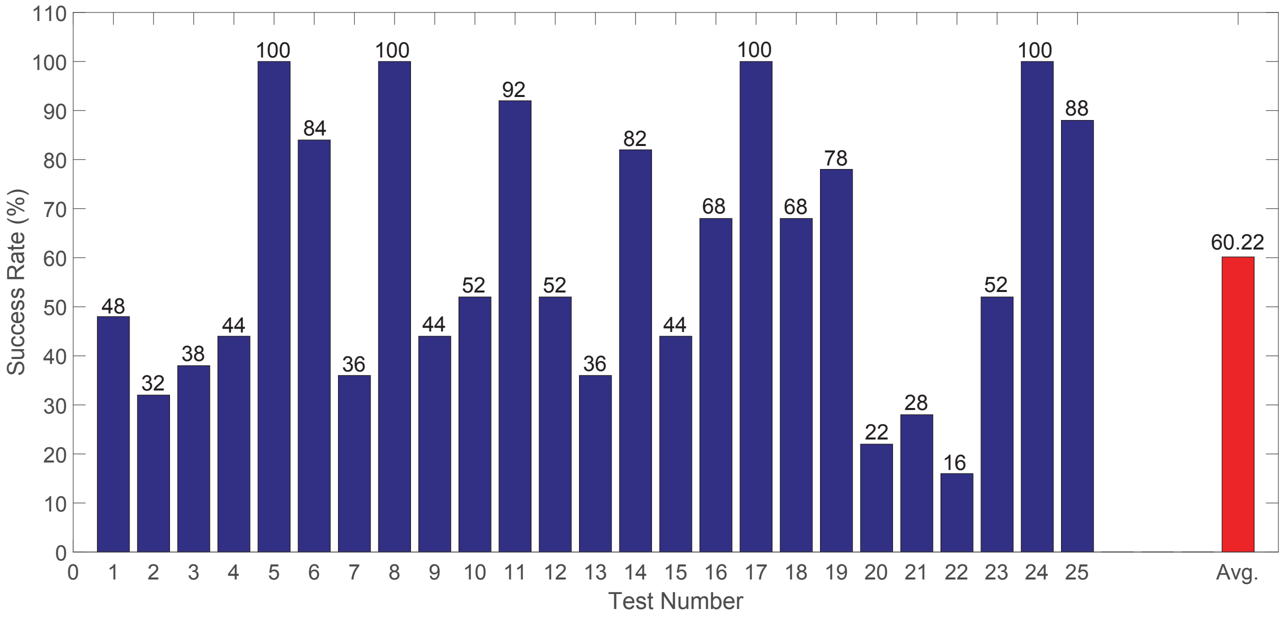

**Figure 13: LLC success rates for different tests in cloud scenario. The blue bar represents the success rate for different tests. The last bar shows the average success rate for all 25 tests**

monitors 320 set-slice pairs since the processor has 10 slices and 32 different set numbers satisfying $s \ mod \ 64 = 0$.

The set-slice pairs are very noisy on the cloud therefore even if the candidate VM is not co-located with the spy VM, there are some active sets in LLC because of the noise from other VMs. However, when the frequency domain of active sets is checked by the spy, there is no dominant frequency component or the dominant frequency components are not consistent with the ping frequency. If the target VM is co-located with the spy VM, then the periodic cache misses can be seen in one of the active sets in Figure 14.

After applying Fourier Transform with an appropriate $F_s$, the dominant frequencies are clearly seen in Figure 15. In order to calculate the frequency domain the sampling frequency $F_s$ should be computed before the frequency transformation is applied to the data. After averaging all LLC sets, $F_s$ is determined to be around 1,800 clock cycle. The normal CPU frequency of the processor is 2.5 GHz so $F_s$ is equal to 1.56 MHz on Amazon EC2 VMs. When the ping requests are sent every 0.4 ms from Spy VM2, then Spy VM1 can monitor the cache misses in active sets as in Figure 14. When the frequency domain is generated, the

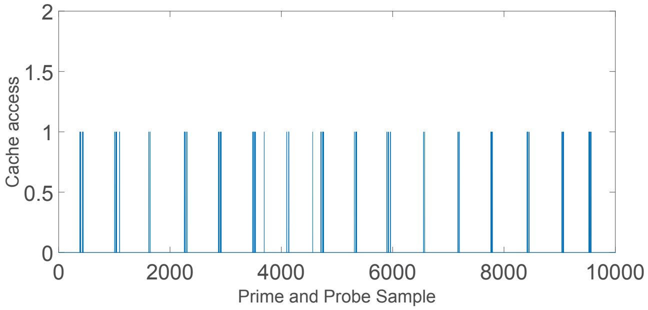

**Figure 14: Cache miss pattern of received ping requests in LLC**

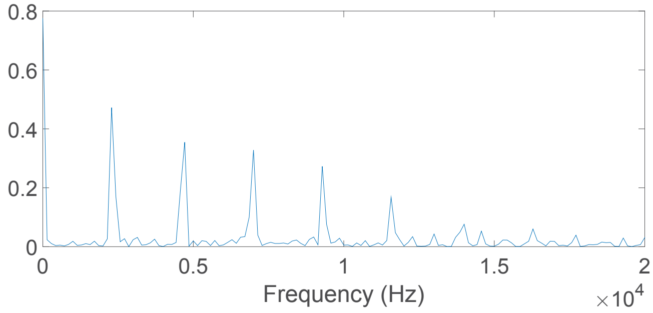

**Figure 15: Frequency components of ping requests in LLC**

frequency components overlap with the frequency of ping requests in Figure 15.

## 5. COUNTERMEASURES

In this section we discuss some details to prevent LLC profiling in the cloud.

**Disabling Huge Pages**: In order to create eviction sets for Prime&Probe profiling Huge Pages should be allocated by the spy. Therefore, if the Huge Pages are disabled by the VMM, then creating eviction sets process becomes harder and the efficiency of the monitoring other VMs drops significantly.

**Private LLC Slices**: Co-located VMs can use whole LLC if they are physically in the same machine. Therefore, if LLC is separated for each VM in the cloud, the profiling phase will be impossible. Nevertheless, this change in the cache architecture is a painful process for cloud providers and it is not efficient.

**Adding Noise to LLC**: One way to avoid LLC profiling is to add noise by flushing some cache lines. Therefore, even if there is no evicted line in a LLC set by the monitored application, the spy assumes there is a cache miss. With additional noise the frequency domain representation of the hit-miss trace will change and the success rate of the ML algorithms will decrease.

**Page Coloring**: Page coloring is a software technique that manages how memory pages are mapped to cache lines. Shi et al. [31] introduced page coloring to partition LLC dynamically to limit cache side channel leakage in multi-tenant clouds. Hence, spy VMs cannot interfere with other VMs in the cloud. However, this method introduces performance overheads and the performance of a program can drastically change between runs.

## 6. CONCLUSION

In this paper we tackled the problem of automating cache attacks using machine learning. Specifically, we devised a technique to extract features from cache access profiles which subsequently are used to train a model using support vector machines. The model is used later for classification applications based on their cache access profiles. This allows, for instance, a cloud instance to spy on applications co-located on the same server. Even further, our technique can be used as a discovery phase of a vulnerable application, to be succeeded by a more sophisticated fine grain attack. We validated our models on test executions of 40 applications bundled in the Phoronix benchmark suite. Using L1 and LLC cache access profiles our trained model achieves classification rates of 98% and 78%, respectively. Even further, our model achieves a 60% (for a suite of 25 applications) in the noisy cross-VM setting on Amazon EC2.

## 7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation, under grants CNS-1618837 and CNS-1314770.

## 8. REFERENCES

[1] AWS IP Address Ranges. https://ip-ranges.amazonaws.com/ip-ranges.json.

[2] Phoronix Test Suite Tests. https://openbenchmarking.org/tests/pts.

[3] Aciiçmez, O., Schindler, W., and Çetin K. Koç. Cache Based Remote Timing Attack on the AES. In *CT-RSA 2007*, pp. 271–286.

[4] Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.

[5] Akata, Z., Perronnin, F., Harchaoui, Z., and Schmid, C. Good practice in large-scale learning for image classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence 36*, 3 (2014), 507–520.

[6] Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., and Butler, K. Detecting Co-residency with Active Traffic Analysis Techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*.

[7] Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[8] Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks against AES. In *CHES 2006*, vol. 4249 of *Springer LNCS*, pp. 201–215.

[9] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 987–1004.

[10] Brumley, B. B., and Hakala, R. M. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security* (2009), Springer, pp. 667–684.

[11] Chandrashekar, G., and Sahin, F. A survey on feature selection methods. *Computers & Electrical Engineering 40*, 1 (2014), 16–28.

[12] Chang, C.-C., and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1–27:27. Software available at http://www.csie.ntu.edu.tw/˜cjlin/libsvm.

[13] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security* (2015), pp. 897–912.

[14] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. *IEEE S&P* (2011), 490–505.

[15] Gulmezoglu, B., Inci, M., Irazoki, G., Eisenbarth, T., and Sunar, B. Cross-vm cache attacks on aes.

[16] Hospodar, G., Gierlichs, B., De Mulder, E., Verbauwhede, I., and Vandewalle, J. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering 1*, 4 (2011), 293–302.

[17] Inci, M. S., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. Co-location Detection on the Cloud. In *COSADE 2016*.

[18] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Tech. rep. http://eprint.iacr.org/.

[19] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *IEEE S&P 2015*.

[20] Irazoqui, G., Eisenbarth, T., and Sunar, B. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on* (2015), IEEE, pp. 629–636.

[21] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID 2014*.

[22] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID 2014*. pp. 299–319.

[23] Lerman, L., Bontempi, G., and Markowitch, O. Side channel attack: an approach based on machine learning. *Center for Advanced Security Research Darmstadt* (2011), 29–41.

[24] Liu, F. and Yarom, Y. and Ge, Q. and Heiser, G. and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE S&P 2015*.

[25] Maurice, C. and Le Scouarnec, N. and Neumann, C. and Heen, O. and Francillon, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters . In *RAID 2015*.

[26] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS 2015*, pp. 1406–1418.

[27] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. CT-RSA 2006.

[28] Pan, Y., Shen, P., and Shen, L. Speech emotion recognition using support vector machine. *International Journal of Smart Home 6*, 2 (2012), 101–108.

[29] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 1–18.

[30] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS '09*, pp. 199–212.

[31] Shi, J., Song, X., Chen, H., and Zang, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on* (2011), IEEE, pp. 194–199.

[32] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS* (2003), pp. 62–76.

[33] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security 2015*.

[34] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *(USENIX Security 2014)*.

[35] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. Mapping the intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. http://eprint.iacr.org/.

[36] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE S&P 2011*.

[37] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS 2012*.

[38] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in PaaS clouds. In *CCS* (2014), pp. 990–1003.

## 9. APPENDIX

| Test Number in Fig. 10,12 | Type | Time(s) | Test Number in Fig. 13 | Description |
|---|---|---|---|---|
| 1) APACHE | System | 61 | 19 | Apache benchmark program |
| 2) DCRAW | PC | 67 | 20 | Convert a RAW image to PPM image |
| 3) FFTW | PC | 292 | 21 | Computes discrete Fourier transform (DFT) |
| 4) GNUPG | PC | 15 | | Encrypts a file using GnuPG |
| 5) HIMENO | PC | 62 | 22 | Implements point-Jacobi method |
| 6) OPENSSL | PC | 22 | | Implements SSL and TSL |
| 7) DOLFYN | PC | 34 | 23 | Implements Computational Fluid Dynamics |
| 8) JAVA-SCIMARK2 | PC | 34 | | Runs Java version of Scimark |
| 9) JOHN-THE-RIPPER | PC | 68 | | Password cracker |
| 10) BOTAN | PC | 401 | | Implements AES-256 |
| 11) ESPEAK | PC | 49 | | Speech synthesizer |
| 12) GMPBENCH | PC | 480 | | Test of the GMP 4.3.0 math library |
| 13) HMMER | PC | 60 | | Implements Hidden Markov Models |
| 14) MAFFT | PC | 39 | | Performs an alignment process |
| 15) GCRYPT | PC | 30 | 25 | Libgcrypt with CAMELLIA256-ECB |
| 16) NPB | PC | 1440 | | High-end computer systems benchmark |
| 17) CLOMP | PC | 48 | 1 | Measures OpenMP overheads |
| 18) BORK | PC | 20 | 2 | Cross-platform encryption utility |
| 19) C-RAY | PC | 147 | 3 | Tests the floating-point CPU performance |
| 20) FFMPEG | PC | 37 | | Tests audio/video encoding performance |
| 21) MINION | PC | 74 | 4 | Constraint solver |
| 22) NERO2D | PC | 650 | 5 | Two-dimensional TM/TE solver |
| 23) NGINX | System | 43 | 6 | Apache program against nginx |
| 24) PERL-BENCHMARK | PC | 88 | 7 | Implementing different versions of perl |
| 25) POSTMARK | Disk | 56 | 8 | Small-file testing in Web and mail servers |
| 26) SMALLPT | PC | 558 | | C++ global illumination renderer |
| 27) STOCKFISH | PC | 6 | | Chess benchmark |
| 28) SUDOKUT | PC | 14 | 9 | Sudoku puzzle solver |
| 29) SYSTEM-LIBXML2 | PC | 96 | 24 | Parse a random XML file with libxml2 |
| 30) VPXENC | PC | 84 | 10 | Video encoding test of Google's libvpx |
| 31) COMPRESS-GZIP | PC | 14 | 11 | Implements Gzip compression |
| 32) CRAFTY | PC | 86 | 12 | Chess engine |
| 33) POLYBENCH-C | PC | 8 | 13 | C-language polyhedral benchmark |
| 34) PRIMESIEVE | PC | 418 | 14 | Generates prime numbers |
| 35) TTSIOD-RENDERER | PC | 168 | 15 | A portable GPL 3D software renderer |
| 36) MENCODER | PC | 31 | | Tests audio/video encoding performance |
| 37) FHOURSTONES | PC | 143 | 16 | Integer benchmark for connect-4 game |
| 38) EBIZZY | PC | 22 | 17 | Generates workloads onweb server workloads |
| 39) HPCC | PC | 12600 | | Cluster focused benchmark |
| 40) N-QUEENS | PC | 260 | 18 | solves the N-queens problem |