Modeling side-channel cache attacks on AES

Samira Briongos
CCS-LSI
Technical University of Madrid, Spain
samira.briongos@die.upm.es

Pedro Malagón
CCS-LSI
Technical University of Madrid, Spain
malagon@die.upm.es

José L. Risco-Martín
Complutense University of Madrid
28040 Madrid, Spain
jlrisco@ucm.es

José M. Moya
CCS-LSI
Technical University of Madrid, Spain
josem@die.upm.es

ABSTRACT

In recent years, side-channel attacks have gained increasing attention, mainly due to their ability to extract sensitive information from their victims in an effortless way. Also, with the development and spread of cloud computing, where victims and potential attackers share physical infrastructure, these attacks are becoming a serious concern. For performance reasons, several resources as CPU cache memories have to be shared, leaving a door opened for attackers. However, when cryptographic processes are properly characterized it is possible to detect attacks which abuse one shared resource as, for example, CPU cache. In this paper we present a timing characterization of a process implementing a cryptographic algorithm such as AES. Then we characterize the same encryption process when suffering a cache attack and when sharing the CPU with other different processes to evaluate how they affect it and get accurate models. The main idea of this work is getting an accurate timing model to distinguish when a process is or not being attacked regarding to timing measurements. Once we get the model, we provide a detection algorithm that detects over 96% of attacks with false positive rates around 5%. The false positive rate is reduced to 0% when discarding the initial transitory state related to the booting stage of a new process.

Author Keywords

Side-Channel Attack; Cache Attack; AES, Attack detection;

ACM Classification Keywords

C.4 PERFORMANCE OF SYSTEMS: *Modelling Techniques*; D.4.6 OPERATING SYSTEMS: Security and Protection—*Invasive software*; D.4.8 OPERATING SYSTEMS: Performance—*Measurements, Stochastic analysis*; G.3 PROBABILITY AND STATISTICS: *Experimental design*

1. INTRODUCTION

Recent advances in technologies have lead to a massive adoption of cloud computing. Cloud providers give customers the illusion of on demand infinite computing capability and resources. Customers rent computing resources instead of buying and maintaining expensive infrastructures. By sharing resources between customers, providers achieve economies of scale, so they can bring resources at lower prices than dedicated infrastructures.

Different Virtual Machines (VMs) from different customers running on the same physical machines are supposed to be running completely isolated. However, this isolation provided by a virtual machine manager (VMM) is only logical, as the VMs are sharing a physical infrastructure. Therefore, cloud computing also introduces new challenging security and privacy risks derived from the resource sharing.

Sensitive information from a "victim" VM can be gained from a different VM with just one requirement: co-residency. In cloud environments attackers do not need to gain access to the attacked machine or to root privileges, they just need to achieve co-residency. Ristenpart et al. [25] first demonstrated that co-residency is achievable and detectable; they also presented some examples of how to exploit co-resesidency.

The usual way to gain information is by causing a interferences in the victim process, and then deduce some information from the effects of these interferences. CPU cache memories are commonly used as a source of information leakage, as they are usually shared among all CPUs. Processes without any trusting requirements have access to the cache memories and can force cache misses or hits to gain the desired information. Consequently, there is a wide variety of Cache-based side channel attacks, with the CPU cache performing a main role. Cache side-channel attacks such as *Flush+reload* [27] and *prime+probe* [21] provide the most fine-grained information, commonly enough for cryptanalysis.

We consider cache-based attacks to the Advanced Encryption Standard (AES), as it the most widely used symmetric block cipher. AES is used in several US government and industry

SummerSim-SCSC 2016 July 24-27 Montreal, Quebec, Canada

applications, and in many other security products all over the world. Several cross-VM Cache side channel attacks on AES have been published [13, 10] They claim that the effect on the target encryption process is minimal and their attacks are hardly detectable by the victim.

The goal of the work presented is to provide mechanisms to the users for the detection of cache side-channel attacks on their VMs. The number of cache hits and cache misses of an encryption process varies substantially when being under a Cache-based attack. Although this information is available through hardware performance counters, the VMs have no access to them. We consider the effect of the attack on the time elapsed during the execution for the detection of the attack. Detection has to be faster than attacks, in order to avoid their success in gaining information.

The main contributions of our work are:

– We measure the effect of cache-based side channel attacks on the performance of a monitored process in a real experimental setup with other processes running.

– We provide a realistic algorithm for the detection of cache side-channel attacks that also works in VM, just considering execution times measurements.

The remaining of this paper is organised as follows. In Section 2, we present the basic concepts on cache-based attacks and previous works on cache-based attacks on cloud infrastructures. In Section 3 we explain the required concepts of cache-based attacks applied to our case of study: AES. Section 4 introduces the experimental setup and presents the proposed methodology for AES process modeling. In Section 5 we present the detection framework. Finally, in Section 6 we draw some conclusions.

2. BACKGROUND AND RELATED WORK

In order to achieve a better understanding of side channel cache-attacks, in this section we are going to give a basic introduction to the concept of CPU caches and shared memory and how they can be exploited to gain information, followed by a description of relevant cache attacks.

2.1 CPU caches

CPU caches are small and fast memories located between the CPU and main memory, specially designed to hide main memory access latencies. They hold a copy of recently used data which will be probably requested by the processor in a short period of time. If the data requested by the processor is not present in the cache (cache miss), this data will be loaded from main memory and stored in the cache. If it is present in the cache (cache hit), there is no need to access main memory.

A set-associative cache is organized as S sets of W lines (also called ways) each holding B bytes. It is common to name them as *W-way set-associative* cache. Given a memory address, the less significant $log_2 B$ bits locate the byte on the line and the previous $log_2 S$ bits do the same for the set. The remaining high-order bits are used as a tag for each line. The tag is used to discern whether a line is already in the cache or not. As main memory is much larger than CPU caches, multiple memory lines map to the same cache set, much more than lines available in a set. When storing data in a fully filled cache set, a line in the set is replaced according to some policy, usually least recently used (LRU).

In inclusive memories, as the ones available in Intel processors, memory is organized in different levels and data in smaller and low level memories (i.e. L1) must also be somewhere in top level memories (i.e. L2 and L3). The last level cache memory, L3, is shared among all cores. Consequently, what a process being executed in a core does related to L3 cache data may have consequences in other core processes. This fact leads to several cache attacks, explained in 2.3.

2.2 Shared memory

Memory is a limited resource whose lack affects negatively to performance. Consequently, operating systems employ mechanisms such as memory sharing to reduce memory utilization. Given that two different processes are using the same libraries, an operating system has two options: loading twice the same libraries into physical memory or loading the libraries only once. The second option implies to map the same physical page into the address spaces of each process.

Another form of shared memory is known as deduplication, which was originally introduced to improve the memory utilization of VMMs. The cloud hypervisor scans the physical memory and recognizes processes that place the same data in memory (pages with identical content). When several pages happen to have the same content, all the mappings to identical pages are redirected to one of them and the other pages are released. However, if any change is performed by any process in the merged pages, memory is duplicated again.

The Linux memory deduplication feature implementation is called KSM (Kernel Same-page Merging) and appeared for the first time in Linux kernel version 2.6.32. KSM is used as a page sharing technique by the Kernel-based Virtual Machine (KVM). KSM scans only potential candidates instead of the whole memory continuously [3].

The deduplication optimization saves memory, allowing more virtual machines to run on the host machine. It provides as well a reduction in power consumption and system cost. Due to deduplication it is possible to run over 50 Windows XP VMx with 1GB of RAM each on a machine with just 16 GB of RAM [1]. In terms of performance, deduplication is an attractive feature for cloud providers, that after several demonstrations of side-channel attacks exploiting page sharing, they are advised to disable.

2.3 Cache attacks

Cache memories, as well as improving performance, create new covert channels that can be exploited to extract sensitive information. The time elapsed when a process tries to access some data is lower when there is a cache hit on the smallest cache. The time increases related to the size of the memory cache with a cache hit. When there is a cache miss, the data has to be retrieved from main memory and the time is significantly higher. Therefore, when a cryptographic process

accesses some data, the execution time depends on the presence (or not) of the data in the cache memory. This timing information can, as a consequence, be exploited to gain information, such as secret keys, of the cryptographic process.

The cache memory was first mentioned as a covert channel to extract sensitive information in 1992 by Hu [11]. Kocher [17] introduced the first theoretical attacks. Kelsey [16] described the possibility of performing attacks based on cache hit ratios. In 2002, Page [22] studied a theoretical example of cache attacks for DES. Tsunoo [26] investigate timing side-channels due to table lookups for DES. In 2004, Bernstein [4] proposed the first time-driven attack on AES, after observing non-constant times when executing cryptographic algorithms. Although the attack he presented was not practical, his correlation attack has been studied extensively. Another attack was proposed by Percival [23] , who suggested that an attacker could determine cache ways occupied by other processes, measuring access times to all ways of a cache set. The reason is that these times are correlated with the number of occupied ways.

Osvik et al.[21] proposed two techniques, which have been widely used later on: *Evict+time* and *Prime+Probe*. These techniques are intended to allow an attacker to determine the cache sets accessed by a victim process. *Evict+time* consists of three steps: first, an encryption is triggered and its execution time measured; second, an attacker evicts some lines; third, the encryption time is measured again. By comparing the second time with the first measure, an attacker can decide whether the cache line was or not accessed. Higher times will be related to the use of the mentioned line. On the other hand, *Prime+probe* consists of two steps: first, attackers fill a cache set with data, and they trigger (or wait for) the victim to perform an encryption. Next, the attackers access their data and measure the time elapsed in order to determine if the data has been evicted by the victim process. If so, the attacker discovers which lines has been used by the victim.

A significantly more powerful attack that exploits shared memory and the completely fair scheduler (CFS) was proposed by Gullasch [9] . The same principles of the attack were later exploited by Yarom and Falkner [27] who named the method *Flush+Reload*. The target of this attack was the L3 cache as it is inclusive and shared between cores. The *Flush+Reload* technique relies on the existence of shared memory and the *cflush* instruction. This instruction allows an attacker to flush the desired line or lines from the cache to main memory. This way, if shared memory is enabled, the attacker can be sure that, if the victim process needs to retrieve the flushed line, it will have to load it from main memory. *Flush+Reload* also works in three states: first, the desired lines from the cache are flushed. Second, the target runs its process or a fragment of it. Finally, the flushed line is accessed measuring the time required to do it. Depending on the reload time, it is decided if the line was, or not, accessed.

Flush+Reload attacks have demonstrated their power against AES T-table based implementations, as did Irazoqui et al. [13], and also detecting cryptographic libraries [14]. An attack also based on the *Flush+Reload* technique was the Cache template attack [8] , where authors rely on the existence of shared memory between the attacking and the attacked processes (a sharing they enforce). Their proposal eliminates the need of knowing the version of the algorithm under attack.

In virtual environments, shared memory is not always available through deduplication. We still have *Prime+Probe* [12, 18] as a working technique to exploit information leakage. To attack the L3 cache in virtualized environments, researchers have overcome complications as not knowing the mappings between physical and virtual memory [2, 20]. Even when shared memory is not an assumption, co-residency is always a requisite.

2.4 Cache attacks detection

To the best of our knowledge, research in detecting side channels attacks in cloud has not been extensively studied yet. The research focus has been on cache side-channels avoidance or obfuscation rather than in detection. Proposed countermeasures include disabling the cflush instruction or the memory sharing functionality. Raj et al [24] suggested cache hierarchy aware core assignment and page coloring to isolate cache interference between VMs. Jin et al. [15] proposed a cache partitioning mechanism in the cloud. They modified the page allocation algorithm to confine the L2 cache usage of each VM running on the same host. Other proposals include limiting the granularity of time counters used in the attacks [19]

Detection is possible based on the idea that any computation task has an impact on the environment. This impact can be measured, for example, reformulating some of the techniques employed when performing an attack to detect such attacks.

Chiappetta et al. [5] obtain useful information for detection from hardware performance counters (HW counters). HW counters are special purpose registers which hold information about CPU events such as clock cycles, cache hits, cache misses, memory accesses, etc. If a process is causing L3 cache misses in other, as *flush+reload* attacks do, we can detect an increase in the amount of total cache misses. The drawback of this approach is that HW counters are not available inside VMs. Cloud clients willing to detect cache misses or CPU usage cannot use HW counters. However, timing measurements are available. With a proper behavioral model of the activity of the encryption process, timing measurements are enough for detection, as we demonstrate on the following sections. Another important advantage of detection is that it is applicable to any behavior inducing an anomalous use of the cache memory.

3. CASE OF STUDY: AES

The AES algorithm serves as a good example both to explain side-channel cache attacks and countermeasures. The T-table-based OpenSSL implementation has been the preferred victim in successful published attacks[13]. It makes use of four lookup tables to perform encryptions and decryptions. The lines of these T-tables used by the encryption process will be available on the cache memory after.

The AES algorithm is a block cipher which consist of a repeated application of a round transformation on the state, denoted S. The number of iteration rounds, N_r, depends on the size of the key: 10 rounds for 128-bits, 12 rounds for 192-bits and 14 rounds for 256-bits. The concrete details of the implementation of AES are not relevant for our purpose, but interested readers are referred to [7]. In the last round of AES, there is an XOR operation between some data of the lookup tables and the last round key. To sum up, each output ciphertext byte can be seen as stated in Eq. 1; k_i is the i byte of the last round key, $Te_{(i+2)\%4}$ is the lookup table corresponding to the byte and s_i is the state byte i from previous round, which is used as index of the lookup table. Gaining information on the data used from the T-tables, with a known output, provides information on the last round key (which is straightforward related to the encryption key).

$$C_i = k_i \oplus Te_{(i+2)\%4}[s_i] \quad (1)$$

With the *flush+reload* technique applied as explained in [13], we are able to flush a line of the cache containing some known data from the T-tables. The data removed from cache can be used in the last round. After flushing the lines involved in the attack, we trigger an encryption and wait until it finished. Then, we reload the removed data again, measuring the time elapsed in accessing it. Time measurements are done using the $rdtsc$ instruction. Low reload times mean the data has been recently used (cache hit) whereas high reload times mean the data was not used (cache miss). Fig. 1 represents a distribution of reload times when performing an attack flushing four cache lines, one for each T-table. It is easy to see which accesses correspond to cache accesses (L1 to L3 cache levels) and which ones correspond to cache misses or main memory accesses. Times over an established threshold, at 200 cycles in our example, represent cache misses. With the information derived from the data accesses and the ciphertext bytes we are able to obtain the secret key after performing around 3000 encryptions in non-virtualized environments and 8000 encryptions in the case of virtualized environments. In order to avoid the attack, we have to detect it before the full key is leaked: seconds. Another consideration we have

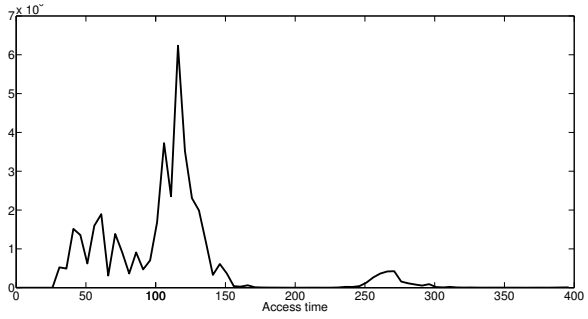

Figure 1: Distribution of times required to read a cache line.

to bear in mind is the probability of not accessing a concrete cache line within an encryption. This probability is given by Eq. 2,where `n` represents the number of table entries a cache line can hold, and N_r the number of rounds of the algorithm.

$$Pr[no\ access\ Te_i] = \left(1 - \frac{n}{256}\right)^{N_r * 4} \quad (2)$$

Each table `Te` is accessed 4 times each round. n depends on the size of the cache line, B. Each value in `Te` has 32-bits, being $n = B/4$. Considering AES-128 ($N_r = 10$) and a cache line of 64 bytes ($n = 16$), the probability of not accessing the cache line is $7,5\%$. It is coherent with the probabilities seen on Fig. 1: the values to the left of the threshold, associated to cache hits at different levels of cache, represent more than 90%. The probability distribution between L1 and L3 cache hits depends on the attacker and victim sharing the same core (sharing L1 cache) or not (sharing L3 cache).

4. MODELING AES

In order to detect cache attacks, first we have to gain knowledge about how an encryption process should behave under different circumstances. We perform several experiments whose aim is to obtain a timing model as accurate as possible. Our experiments are performed using the AES T-table implementation of OpenSSL, version 1.0.1f, in a machine with an Intel Core i7-4790(3.60 GHz) processor with Centos7 OS. The experiments in virtualized environment are performed in an Intel Xeon CPU E3-1226 v3 (3.30GHz) also running Centos, belonging to an openstack cluster with KVM as hypervisor and the deduplication feature enabled. Both machines have a L3 cache of 8MB with a line size of 64 bytes. In each of the considered cases we perform 1 million encryptions.

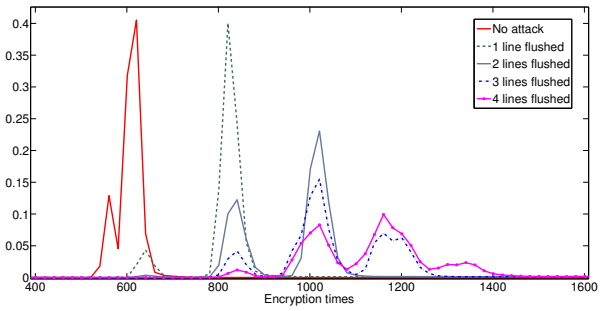

Figure 2: Distribution of times required for encryptions.

It is mandatory to get information of the four T-tables to obtain the full key. However, to perform a successful attack the attacker does not have to flush four lines each time an encryption is performed. We consider, as representatives for detection, the cases in which the attacker flushes from one to four lines per encryption. Fig. 2 depicts the distribution of time elapsed in different cases, for the i7 processor: when performing no attacks and when flushing different number of cache lines. It makes sense that the encryption lasts around 600 cycles, as the encryption of the AES has ten rounds accessing data an each access to L1 cache needs about 50 cycles. Although each round each table is accessed 4 times, the processor performs several reads "simultaneously". Also the distance between peaks in the figure is consistent with the main memory access time. The time distribution matches the probability of cache misses expected from a theoretical probabilistic analysis. Table 1 shows the probability of cache misses (columns) depending on the number of cache lines flushed (rows). Column T shows the theoretical estimation and column E the experimental result, obtained from data shown in Fig. 2.

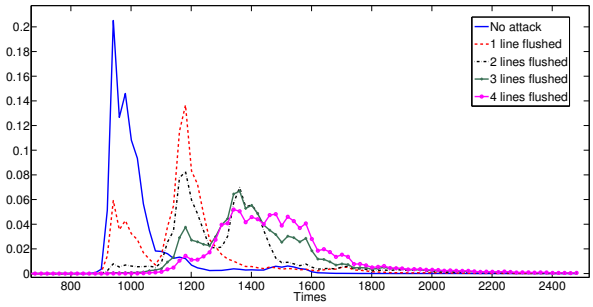

Figure 3: Time distribution of encryption in cloud.

N	Amount of Cache misses (penalty)									
	0		1		2		3		4	
	T	E	T	E	T	E	T	E	T	E
1	7.6	8.6	92.4	85.9	0.0	1.0	0.0	1.0	0.0	3.4
2	0.6	0.8	26.7	32.6	72.7	59.5	0.0	1.5	0.0	5.6
3	0.0	0.0	6.7	10.4	45.3	51.3	48.0	31.9	0.0	6.4
4	0.0	0.0	1.7	3.2	21.3	32.6	50.9	44.9	26.1	19.3

Table 1: Probability distribution of cache misses (%)

From Fig. 2 we can conclude that, as long as there is no other process executing in the CPU, times above 700 cycles are highly suspicious of causing intended delay.

The same experiments are performed between two different co-resident VMs. We obtain the time elapsed distribution, depicted in Fig. 3. The first conclusion is that the threshold established for detecting suspicious scenarios depends on the machine executing the process. The second is that in cloud the peaks of the time distributions are wider than those of non-virtualized processes; distinguishing between distributions is much harder, specially when several lines are flushed. In this case a threshold of about 1100 cycles could work. However, as some non-attack encryption times are above this threshold, a new parameter representing confidence on the detection should be defined to avoid false positives.

4.1 Adding noise

A process performing an encryption is likely to share the CPU with other processes. This fact is more remarkable in cloud environments, considering each VM share computing resources in order to achieve higher CPU utilizations and economies of scale. With the following experiments we want to model how this sharing affects the encryption process, with and without an attacking process, to refine our timing model. The goal is to be able to detect the attacks under different conditions. Our first approach consists in launching the *Lookbusy* program and, at the same time, perform the encryptions with and without the attack. *Lookbusy*[1] stresses, with a synthetic load, different CPU hardware threads to a certain utilization avoiding memory or disk usage. We select different utilization levels and run our experiments.

On Fig. 4 we observe the effect of sharing the CPU with other CPU consuming processes. As the use of CPU increases, the height of the peak representing "non attack" decreases, and other peaks rise up. The height of these peaks are important,

[1]http://www.devin.com/lookbusy/

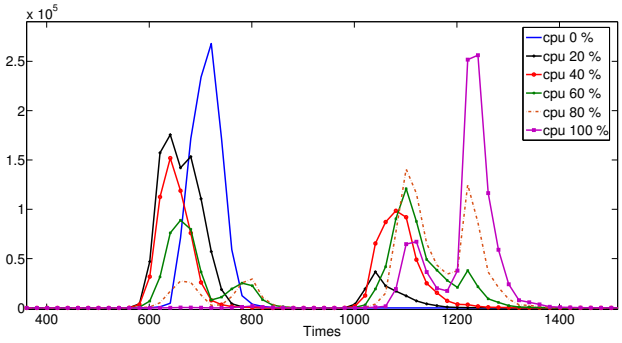

Figure 4: Encryptions time distributions with different CPU utilization levels.

as it is linear with the CPU use. Fig. 5 shows clearer this effect. Seeing the mentioned effect in other way, we can gain information about the CPU use by observing and analysing our own execution times of a modeled process.

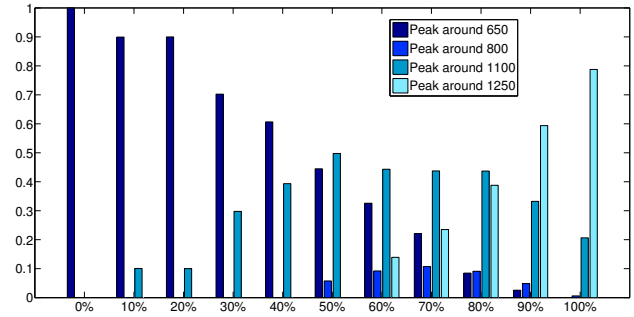

Figure 5: Distribution of times around peaks for different CPU utilization levels.

We consider the use of memory to be a possible interference. We have adapted *RandMem*[2] to access random memory regions individually. We launch *RandMem* simultaneously to the encryptions, accessing memory regions using a random pattern. We perform multiple experiments with different memory size to stress. We measure encryption time under these circumstances. The execution of *RandMem* also consumes CPU. Fig. 6 represents the effect of memory consumption on the encryption times.

In the case of memory consuming processes we also want to check if the delays measured are due to cache misses or not. Consequently, as performance counters are available for non-virtualized processes, we make use of them and measure the number of cache misses per encryption. It turns out that the memory consuming processes were only able to cause at maximum one cache miss per encryption and less than 1% of the encryptions were affected. That is, even when we are making use of lots of memory, the interference in cache between the encrypting and the memory consuming processes is insignificant unless intentionally caused. CPU consumption is much more significant on the time distribution.

[2]http://www.roylongbottom.org.uk

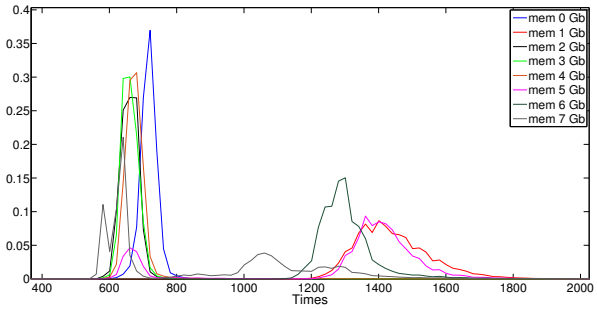

Figure 6: Encryptions time distributions with different memory utilization levels.

As for cloud environments we launch instances of VM consuming CPU, memory or executing the Yahoo! Cloud Serving Benchmark (YCSB) which is used to simulate workloads that fetch web pages, including the writing portions of those workloads [6] as this are realistic loads in cloud. Again the results of the experiments can be seen in Fig. 7, altogether with the peaks distributions in Fig. 8

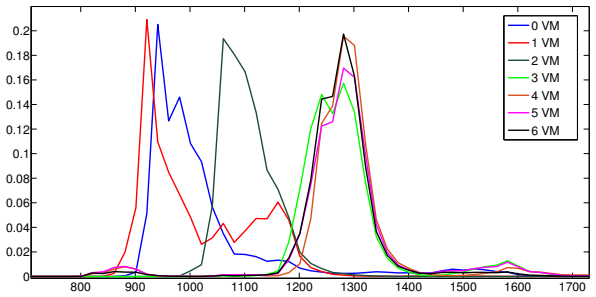

Figure 7: Encryptions time distributions with co-resident virtual machines.

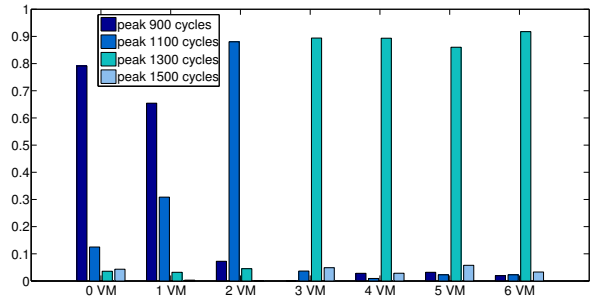

Figure 8: Distribution of times around peaks with co-resident virtual machines.

Considering virtual environments, it is also possible to approximately determine the CPU use of the host in which the process is being executed. During all the execution of the processes we also measure the host machine use. Each launched instance consumes about 25% of the CPU. With 3 instances launched to set CPU usage context and two more VMs, used as server and attacker in our simulation, the CPU utilization is about 100%. With more than three running instances, CPU utilization is about 100% and it slightly decreases when executing our processes. In terms of timings there are no significant differences between the different types of instances; the most relevant parameter is CPU consumption.

Other important requisite we need for detection is that, even in virtualized environments and with a 100% CPU usage, the distributions for each of the considered conditions are still distinguishable. Fig. 9 shows the time distributions for the different conditions: under no attack or flushing from 1 to 4 cache lines.

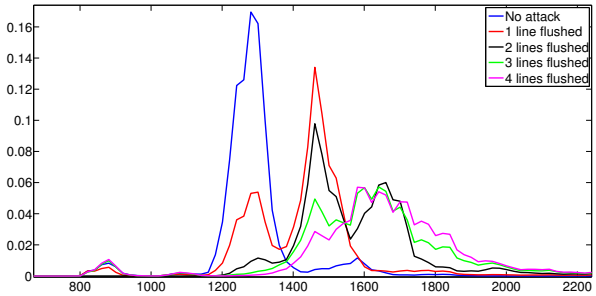

Figure 9: Distribution of times for encryptions in virtualized environments 100% CPU usage.

5. PROPOSED DETECTION ALGORITHM

Based on the models of behavior presented in the previous section we are able to provide a simple detection algorithm 1. The proposed algorithm decides whether the system is under attack or not based on the time distribution measured. It makes use of the previous 200 samples before the current one. Then, we make an histogram selecting time intervals of 20 cycles. Once we have the histogram, we apply a rectangular window of three samples; i.e. it contains the previous, current and next samples. We use the result of windowing the histogram to detect where are the peaks of our distribution. If more than one peak is detected, we use the heights of the peaks to decide if we are being attacked or not. For example, according to the previous models, if the second peak is higher than the first one we will categorize this process as an attack. In this case we do not take further considerations, as checking if the first peak is over a certain value or checking if the two peaks are in fact the same by looking where are placed the peaks. Therefore, we expect some false positives to appear. We define a new parameter, a "confidence", to reduce the number of false positives we get. If a tendency defined as attacking keeps on repeating, the confidence increases until it reaches a threshold; then, we declare it as an an attack, otherwise it decreases.

To validate our detection system we define a test set, different from the training set used in Section 4. We have simulated multiple encryptions with different conditions: under attacks on different number of lines, under no attack in both virtualized and non-virtualized environments. The rates of false positives and false negatives are shown in tables 2 and 3 for different number of lines flushed, where 0 lines flushed means no attack. Each process performs 8000 encryption requests in

Algorithm 1 Cache attack detection algorithm

Input: **Encryption times, confidence level**
Output: *flag* ***detected***

```
h = calculate_hist (times_{n-200...n});
w = window (h);
arr_peaks = peak_search (w);
if length (arr_peaks) > 1 then
    if arr_peaks [0] < arr_peaks [1] then
        confidence + +;
    else
        if (num_significant_peaks ≠ p_0) >= 2 then
            confidence + +;
        else
            confidence − −;
        end if
    end if
else
    confidence − −;
end if
if confidence > threshold then
    detected = 1;
end if
return confidence, detected;
```

non virtualized environments or 10000 in virtualized environments. We consider a false positive if a non-attacking process is labelled as an "attack"; on the other hand, a false negative is an attack we do not detect in time.

Flushed lines	Non-virtual environment	Virtual environment
0	4%	4.8%

Table 2: False positives rates(%)

Flushed lines	Non-virtual environment	Virtual environment
1	2%	3.6%
2	1.1%	2%
3	0.5%	2%
4	0.4%	1.5%

Table 3: False negative rates(%)

Even that our detection algorithm is quite simple, it is able to detect over 96% of the attacks. However, the experiments performed have shown that, for higher CPU utilizations, false positive rates increase as the distributions are more similar to the attacking ones. We are not considering this fact properly in this simple algorithm. We have also detected that there are more false positives when the encryption process has just begun and the times have not jet stabilized, in transient periods.

The proposed algorithm provides a proof of the utility of the models performed in Section 4. The detection algorithm can be improved by adding information on the evolution of times or the CPU usage. Other alternatives include improvements on the peak extraction algorithm, in order to label two nearby peaks as one or increasing the number of samples of the histogram. Moreover, we believe that machine learning techniques, such as neural networks or predictive algorithms, can be applied using our models to perform detection. The idea would be measuring how different is the real behavior from the expected behavior and how similar it is to attacks in order to decide whether we are being attacked or not.

When an attack is detected, stopping the encryptions provides information to the attacker, so it can react and scape. We suggest the use of a fake encryption key when detecting an attack and keep on performing encryptions. We will be able to identify the identifier of the process responsible of the attack. In cloud environments, we are not able to kill other clients machines, or even to decide where our machine is placed. The fake key is useful to keep the attacker busy and away from our real key while we notify the cloud provider that we are suffering an attack from a co-resident VM. It is in hands of the cloud provider to detect the attacking VM and take the actions they consider. We are giving to the provider information about a suspicious VM that can be attacking other VMs from other cloud clients who will probably be grateful for the protection. However, in order to report an attack to the provider, certainty is required in order not to defame other clients. Even when the provider does nothing, the attacker will probably stop the attack when he gets the fake key. In order to avoid being detected, it might end the attack. Therefore, once the encryption process comes to its normal state, we can keep on encrypting data with the real key.

6. CONCLUSION

From our point of view attack detection is a powerful tool that works better than disabling options that enhance performance as memory sharing. When a door useful for attacks closes, a window that can be exploited opens. If you know what should be happening inside the "house" and you detect that something different is happening, you are alerted and you can defend yourself from whatever comes through the window.

In this work we have provided useful information about how cache side channel attacks work, and we have characterized a process performing AES encryptions being attacked and performing normally. We have refined our models considering the encryption process at different circumstances such as different CPU loads or memory use. With all this information we have provided a simple detection algorithm for AES attack that can be easily simulated and extended to any other cyptographic algorithm susceptible of a cache attack. We have characterized all the processes in terms of timing, as time measuring is available inside VMs in contrast of performance counters which are not.

Even with a quite simple detection algorithm, when the monitored process is properly modeled, we are able to detect over 96% of the different attacks with reasonable false positive rates even in cloud environments. The false positives appear during the initial transitory state of a new process, thus if this state is discarded by the algorithm or the measurements are taken when the monitored system is stable, the false positive rate tends to 0%. We have demonstrated that detecting cache attacks with only time measurements is possible, as long as we can model how our process should work. We do not need to be the hypervisor in cloud environments in order to protect ourselves. There is still room for improvement on the detection algorithms, once that they have demonstrated their utility for cache attacks detection.

ACKNOWLEDGMENTS

This project has been partially supported by the Spanish Ministry of Economy and Competitiveness, under contracts TEC-2012-33892, and RTC-2014-2717-3.

REFERENCES

1. Kernel samepage merging. http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb/, 2015.
2. Apecechea, G. I., et al. Systematic reverse engineering of cache slice selection in intel processors. *IACR Cryptology ePrint Archive 2015* (2015), 690.
3. Arcangeli, A., et al. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium* (July 2009), 19–28.
4. Bernstein, D. J. Cache-timing attacks on aes. Tech. rep., 2005.
5. Chiappetta, M., et al. Real time detection of cache-based side-channel attacks using hardware performance counters. *IACR Cryptology ePrint Archive 2015* (2015), 1034.
6. Cooper, B. F., et al. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, ACM (New York, NY, USA, 2010), 143–154.
7. Daemen, J., and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.
8. Gruss, D., et al. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), 897–912.
9. Gullasch, D., et al. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium S&P 2011, 22-25 May 2011, Berkeley, California, USA* (2011), 490–505.
10. Gülmezoglu, B., et al. A faster and more realistic flush+reload attack on AES. In *6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015.* (2015), 111–126.
11. Hu, W. Lattice scheduling and covert channels. In *1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 4-6, 1992* (1992), 52–61.
12. Inci, M. S., et al. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Tech. rep., IACR Cryptology ePrint Archive, 2015.
13. Irazoqui, G., et al. Wait a minute! A fast, cross-vm attack on AES. In *17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings* (2014), 299–319.
14. Irazoqui, G., et al. Know thy neighbor: Crypto library detection in cloud. *PoPETs 2015*, 1 (2015), 25–40.
15. Jin, X., et al. A simple cache partitioning approach in a virtualized environment. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications* (Aug 2009), 519–524.
16. Kelsey, J., et al. Side channel cryptanalysis of product ciphers. *Journal of Computer Security 8*, 2/3 (2000), 141–158.
17. Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings* (1996), 104–113.
18. Liu, F., et al. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium S&P 2015, San Jose, CA, USA, May 17-21, 2015* (2015), 605–622.
19. Martin, R., et al. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA* (2012), 118–129.
20. Maurice, C., et al. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings* (2015), 48–65.
21. Osvik, D. A., et al. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings* (2006), 1–20.
22. Page, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive 2002* (2002), 169.
23. Percival, C. Cache missing for fun and profit. In *Proc. of BSDCan 2005* (2005).
24. Raj, H., et al. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, ACM (New York, NY, USA, 2009), 77–84.
25. Ristenpart, T., et al. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference CCS'09, Chicago, Illinois, USA, November 9-13, 2009* (2009), 199–212.
26. Tsunoo, Y., et al. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 5th International Workshop CHES 2003, Cologne, Germany, September 8-10, 2003* (2003), 62–76.
27. Yarom, Y., and Falkner, K. 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.* (2014), 719–732.