This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems

# Cross-VM Cache Attacks on AES

Berk Gulmezoglu, Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth and Berk Sunar

***Abstract*—Cache based attacks can overcome software-level isolation techniques to recover cryptographic keys across VM-boundaries. Therefore, cache attacks are believed to pose a serious threat to public clouds. In this work, we investigate the effectiveness of cache attacks in such scenarios. Specifically, we apply the *Flush+Reload* and *Prime+Probe* methods to mount cache side-channel attacks on a popular `OpenSSL` implementation of AES. The attacks work across cores in the cross-VM setting and succeeds to recover the full encryption keys in a short time—suggesting a practical threat to real-life systems. Our results show that there is strong information leakage through cache in virtualized systems and the software implementations of AES must be approached with caution. Indeed, for the first time we demonstrate the effectiveness of the attack across co-located instances on the Amazon EC2 cloud. We argue that for secure usage of world's most commonly used block cipher such as AES, one should rely on secure, constant-time hardware implementations offered by CPU vendors.**

***Index Terms*—Cross-VM Side-Channel Attacks, Cache Attacks, Memory De-duplication, Prime+Probe, Flush&Reload.**

## I. Introduction

In recent years, cloud and virtualization adoption by both government and private sector has reached unprecedented levels. From game servers to mobile application databases, more and more of the previously privately owned and managed systems are moving to cloud. To take advantage of this opportunity major tech companies such as Amazon, Google and Microsoft have become Cloud Service Providers (CSP) while Netflix, Dropbox, Instagram, Pinterest have became some of their biggest customers. However, the full potential of the cloud is still not realized. The biggest concern preventing companies and individuals from further adopting the cloud is data security and personal privacy.

In virtualized systems, multiple users share the underlying hardware for better utilization and lower cost. On the other hand, these users do not necessarily know or trust each other and require strong isolation. Sandboxing is enforced by the hypervisor to realize this in a secure and efficient way. However, sandboxing has traditionally been defined in the software space, thus, ignoring leakages of information through subtle side-channels shared by the processes running on the same physical hardware. This is partly due to the fact that, traditionally software libraries and applications designed for servers were implemented assuming trusted servers and/or neighbor VMs. For privacy critical data, especially cryptographic data, this gives rise to a blind spot where isolation can break. Even though classical implementation attacks targeting cryptosystems have been studied extensively, so far, there has been little discussion about safe implementation of cryptosystems on cloud systems where multiple tenants run on the same CPU. Indeed, given the level of access to the server and control of process execution required to realize a successful physical attack, these attacks have been largely dismissed in the cloud setting until fairly recently.

It was in 2009 when it was first demonstrated that it is possible to solve the logistics problems and extract sensitive data across VMs. Specifically, using the Amazon EC2 service as a case study, it was demonstrated that it is possible to identify where a particular target VM is likely to reside, and then instantiated new VMs until one becomes co-resident with the target VM. By solving the co-location problem, this initial result finally established a viable and realistic scenario where microarchitectural side channel attacks can be employed to steal secrets.

Later on, new and more powerful covert channels were exploited in virtualized environments. In fact, microarchitectural attacks have shown to be able to recover a wide variety of private information, e.g., from cryptographic keys to sensitive information from a co-located user's shopping cart. In general, the source of these vulnerabilities is that people are running software and code that was designed for single user machines/privately owned servers instead on shared cloud systems. However, as explained above, when sharing physical resources, one has to be wary of his neighbors and consider them as potentially malicious parties.

**Our Contribution:** In this work, we present a novel cache based attack on the AES cipher. In contrast to previous works [1], [2] we use the LLC as a covert channel, which is shared across cores in modern processors. This allows us to recover an AES key even when the victim

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



and spy are not co-residing in the same core.

We utilize two novel side channel techniques, i.e., *Flush+Reload* and *Prime+Probe* applied in the LLC. Whereas the first one requires memory de-duplication features to be activated at the hypervisor level, the latter does not need special requirements to succeed. We demonstrate the viability of the attack by running in different processors and hypervisors, e.g., VMware for *Flush+Reload* (since they both implement de-duplication) and VMware and Xen for *Prime+Probe* (without de-duplication). Finally we demonstrate the viability of the *Prime+Probe* attack in the real cloud by running it on Amazon's EC2 servers.

In short, this paper:

- Introduces two spy processes that use the LLC as a covert channel;
- Discusses the differences and applicability of spy processes in both native and cross-VM scenarios;
- Presents a new key recovery method for a T-table based AES algorithm and the first cross-VM LLC attack on AES recovering the full key;
- Introduces the **first attack on AES in a public cloud**, i.e., the Amazon EC2 cloud; and
- Presents a thorough comparison of the effectiveness and applicability of the *Flush+Reload* and *Prime+Probe* attacks and techniques based on distinguishers in both the native and cross-VM scenarios including Amazon's EC2.

The rest of the paper is organized as follows: first we present the background to understand the attacks in Section II, then we explain our attack and measurement procedure in Sections III and IV, then we present our results in the different scenarios in Section V and finally we point out the conclusions in Section VII.

## II. Background

In this section we give a brief overview of the related work in terms of cache attacks and we explain the necessary background to understand the attacks we performed.

### A. Related Work

In recent years, cache attacks have become more practical and popular due to the widespread adoption of virtualized systems and cloud computing. Cache attacks were first considered as a possible covert channel as early as 1992 by Hu [3]. Even though the work served as an initial theoretical study on cache attacks, it was not until 2000 when Kelsey et al. [4] expanded it by distinguishing cache hit/miss ratios. Shortly later, theoretical models of cache attacks were investigated by Page [5] whereas Tsunoo et al. [6] proposed the first cache based attacks against DES.

The first practical implementations of cache attacks on AES were proposed in 2004. Bernstein [7] recovered a full AES key by implementing a timing attack based on micro-architectural timing differences when the memory blocks are loaded in different positions of the cache. Around the same time, Osvik et al. [2] studied the viability of two novel cache based spy processes, i.e., *evict + time* and *Prime+Probe*. The first one works by observing the difference between two equal encryptions when a memory block in the cache is evicted in between the encryptions. If the memory block is used by the encryption algorithm, it will take less time to complete the second encryption since the used data will be already loaded to the cache. The latter technique fills the cache with attacker's own data before the encryption, and checks after the encryption which memory blocks still remain in the cache, i.e., which memory blocks have not been used by the algorithm. While both methods achieve full key recovery, *Prime+Probe* requires less traces.

In the next couple of years, variants of the above mentioned spy processes were studied against different cryptographic algorithms. Bonneau and Mironov exploited a variant of Bernstein's attack, but targeting cache collisions in the last round of AES [8]. Later, Acıiçmez et al. showed that collisions in the first and the second AES rounds could also be exploited to retrieve an AES key [9]. Improvements over [1] were also studied by Neve and Seifert, applying a similar spy process to the last round of AES. Later, Acıiçmez demonstrated that cache spy processes are also effective against public key cryptographic algorithms like RSA [10].

Despite the progress that was achieved in terms of side channel attacks, they were still believed to have low practicality due to the difficulty of the scenario (two processes running under the same Operating System) in which they were applied. However by 2009 cloud computation and virtualized environments were popular enough to start considering the implications of cache side channel attacks across co-located Virtual Machines (VMs). In fact, Ristenpart et al. [11] were able to detect co-located VMs in the Amazon EC2 public cloud, and deduce key strokes from the co-located victim by monitoring the cache.

In the following years, researchers kept on exploring the scenario presented in [11]. Indeed, only one year later Zhang et al. [12] presented a new method to detect whether any tenant is co-located in virtualized environments. The method was based on monitoring and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



detecting whether anyone else was using the upper level caches. Shortly later, again Zhang et al. [13] recovered the ElGamal decryption key using the above explained *Prime+Probe* attack from a core co-located VM. Around the same time, a new cache attack that would later acquire the name of *Flush+Reload* was proposed by Gullasch et al. [14]. With the new spy process, authors were able to recover an AES key in native OS scenario by controlling the Completely Fair Scheduler (CFS).

The previously proposed attacks succeeded in scenarios where victim and attacker are co-located in the same CPU core and share L1 and L2 caches. However, with multicore computers and servers becoming more popular, the difficulty of achieving this co-residency increased. Yarom et al. [15] were the first ones overcoming this issue, by using a covert channel that is shared by all the cores, i.e., the Last Level Cache (LLC). Using the *Flush+Reload* spy process, they were able to recover a full RSA key across VMs residing in different cores, thereby increasing the practicality (and popularity) of cache based side channel attacks. Shortly later, Irazoqui et al. [16] used the same method to mount a new cache attack by monitoring last round of an AES encryption. Both attacks demonstrated that the sandboxing techniques implemented by various hypervisors (e.g., Xen, KVM and VMware) could in fact be bypassed.

The *Flush+Reload* attack was also applied in many other attack scenarios. Namely, Platform as a Service clouds in [17], security protocols and improper patches [18], Eliptic Curve Cryptography (ECC) [19] and building effective cache template attacks [20]. However, one of the main disadvantages of the *Flush+Reload* spy process is that it requires de-duplication features to be enabled by the hypervisor. Although this might be popular in lab based environments, de-duplication is usually not enabled in commercial IaaS clouds. However, Liu et al. and Irazoqui et al. demonstrated in concurrent works that LLC attacks were also possible without de-duplication by recovering ElGamal and AES keys across VMs respectively in [21], [22]. They implemented the well known *Prime+Probe* attack in the LLC, overcoming difficulties such as sliced caches and virtual to physical address mappings. These works were later expanded by Oren et al. [23], implementing the *Prime+Probe* in javascript to infer the websites visited by potential victims. Recently, Inci et al. [24] implemented the first key recovery attack in the Amazon EC2 public cloud across co-located VMs.

### *B. Cache addressing and virtual-physical memory mapping*

Modern processors use virtual memory to protect processes from accessing directly the physical memory. The OS takes care of translating these virtual addresses to their physical mapping. The physical memory in most modern systems is divided into *memory pages* of 4KB size. The page size plays a crucial role in the translation stage, since the number of bits from the virtual address that have to be translated directly depends on it. Indeed, if $p_o$ is the page size in bytes, the lower $log_2(p_o)$ bits of the virtual address will not be translated by the Memory Management Unit (MMU) and will remain the same in both the physical and virtual addresses. This is what we call the *page offset*. The rest of the bits will be refereed as the *virtual page frame number* before the translation and the *page frame number* after the translation.

In order to efficiently perform this translation, modern processors have several levels of Translation Lookaside Buffers (TLBs). The TLB is a special cache holding the most recently fetched memory pages and their corresponding virtual page frame numbers. This allows the system to first check the TLB for the requested page translation, speeding up the page lookups.

Aimed at more efficient paging, most processors also allow *hugepage* allocations. Hugepages are substantially larger than regular pages and usually have a separate TLB. As a result of this, a hugepage holds 2 MB of data while occupying a single TLB entry in contrast to 512 entries would be needed with regular pages which in turn reduces the number of TLB misses.

**Memory Addressing in Cache:** There are three widely used cache types: direct mapped (each memory block can only go to one fixed location in the cache), fully associative (a memory block can reside in any position in the cache) and set associative (a memory block can reside in a subset of locations in the cache). We will mainly focus on set associative caches since they are the most common choice in modern processors. Set associative caches are defined by 3 main parameters: the cache size $s$, the cache line length $l$ and the number of ways $w$ for each cache set. Using these parameters, one can calculate the number of sets in the cache as:

$n_s = s/(w * l)$

As Figure 1 shows, each of the memory blocks ($l$ size blocks) will reside in a specific set in the cache, mainly defined by its physical address. For the address translation, the physical address ($pfn+p_o$) is divided into three different categories. The lowest $log2(l)$ bits points

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



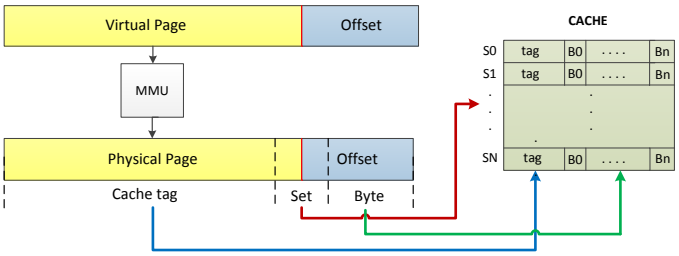



Fig. 1. Cache accesses when it is physically addressed.

to a specific location in a cache line. The following $log2(n_s)$ bits indicates the set in which the data resides. The rest of the bits acts as tag that is used for correct matching of the corresponding memory block.

### C. Memory De-duplication

Memory de-duplication is an optimization technique developed to increase memory utilization and efficiency of virtualized systems. While the de-duplication was originally designed for hypervisors, it was later integrated to non-virtualized systems as well. De-duplication works by scanning system memory for duplicate entries, commonly found in virtualized systems and merge these entries to save memory. After the detection of duplicate entries, multiple copies are cleared from memory and a single copy is shared between users.

While the mechanism is useful in OS memory management for merging shared libraries used by different applications, it is even more beneficial in virtualized systems where many VMs run the same OS and/or software. In fact, in [25], researchers were able to run 52 Windows XP guest VMs with 1 GB of RAM each, on a system with only 16 GBs of physical RAM.

Note that while the specific implementations of de-duplication may differ on parameters such as scan interval, block size and area, the end result is the same. As an example, we will detail the Kernel Samepage Merging (KSM) used by the Kernel-based Virtual Machine's (KVM) [26], [25]. It is introduced to the Linux kernel in version 2.6.32 [27] and indirectly in KVM hypervisor. KSM works by *madvise* system call advising the `ksmd` to scan an unshared portion of the memory. Since it would be CPU intensive and time consuming to scan the whole memory, only potential candidates are scanned. During the scan, signatures are created for these pages and added to the de-duplication table. To create signatures, the KSM scans the memory at 20 msec intervals and scans only a portion of the potentially duplicate memory pages at a time. This is the reason

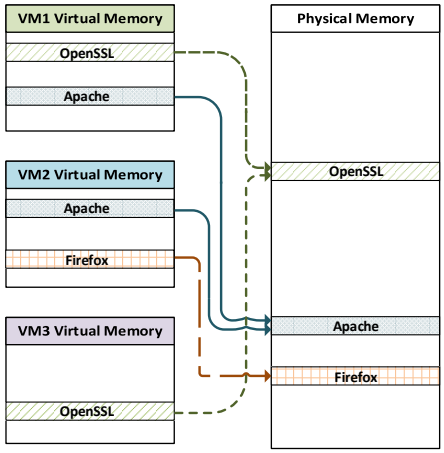



Fig. 2. Memory De-duplication Scheme

why memory disclosure attacks like [28] has to wait for a certain time before the de-duplication takes effect and only then the attack can be performed. During the memory search, the KSM analyzes three types of memory pages [29];

- **Volatile Pages**: Pages where the contents of the memory change frequently and should not be considered as a candidate for memory sharing.
- **Unshared Pages**: Candidate pages that the *madvise* system call advises to the `ksmd` to be likely candidates for merging.
- **Shared Pages**: De-duplicated pages that are shared between users or processes.

When pages with matching signatures are found, they are cross-checked to determine if they are identical. If they match, the pages are merged and tagged as copy-on-write (COW). When a process wants to make a change in the de-duplicated page, a copy of the page is created and the changes are applied to this copy, hence terminating the de-duplication.

### D. The Flush+Reload Side-Channel Attack

The *Flush+Reload* attack is a trace driven cache side-channel attack first introduced in [14] and acquired its name in [15]. The attack exploits the shared memory leakage on de-duplication processes. One of the advantages of *Flush+Reload* attack is that the attacker does not need to share core with the victim as long as a shared last level cache exists. There are three main stages to implement the attack:

- **Flush stage:** The first step of the attack involves flushing the desired memory locations from the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



entire cache hierarchy using the `clflush` command. Due to the inclusiveness of the LLC in Intel processors, the `clflush` command will remove the memory block from all cache levels.
- **Victim access stage:** In this stage, the attacker waits for the victim to execute the targeted process.
- **Reloading stage:** The attacker measures the reload time of the previously flushed memory block. If the victim accessed the memory block, it will reside in the cache therefore resulting in a lower reload time. If not, then the targeted memory block resides in the memory, thus resulting in a higher reload time.

### *E. The Prime+Probe Side-Channel Attack*

The *Prime+Probe* attack is a cache based spy process first described 10 years ago by Osvik et al. [2]. The attack procedure consists of three main stages:

- **Cache priming** : In the priming stage, the attacker fills the whole cache with his own memory blocks.
- **Waiting for victims accesses** : In the second step, the attacker waits enough time to let the victim use the previously primed cache. Obviously, some of the primed memory blocks will be evicted by the victim at this step.
- **Probing the primed blocks** : In this stage, the attacker loads the previously primed memory blocks. Some of the blocks (the ones that the victim did not evict) will still reside in the cache, while the other ones will have been evicted to a lower level cache or the memory. This can be noticeable by measuring the loading time for each of the blocks, since lower level cache accesses will be retrieved slower.

In Prime+Probe attack, we need to know which set is used by T-table entries. If the machine has non-linear selection algorithm (which is the challenging part of the attack), then we can find the T-table positions in the LLC through Algorithm 1. After we get the location (which set and slice in the LLC) of the T-tables in the LLC, we need to fill that set of the LLC with our data. After a while if the victim uses this specific T-table, the loading time is higher. The loading means reading all our data and measure the timing the reading of 20 (set-associativity) lines. If there is an access to T-table, the loading time is higher and we can conclude that the T-table is used in that specific encryption.

Although *Prime+Probe* has been known for many years, it was not until one year ago when it was applied to the LLC. Some of the reasons why it was not trivial to modify the *Prime+Probe* attack to the LLC are:

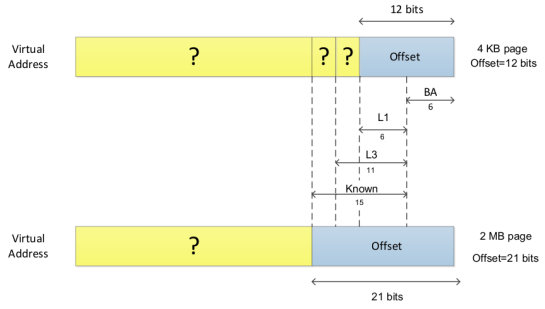



Fig. 3. Regular Page (4 KB, top) and Hugepage (2MB , bottom) virtual to physical address mapping for an Intel x86 processor. For Hugepages the all L3 cache sets are transparently accessible even with virtual addressing.

- **Large cache**: The LLC is usually in the order of MBs making it impractical to prime the whole set.
- **Unknown physical bits**: Due to larger size of the LLC, the location of the memory blocks in the LLC is unknown. With small caches (like the L1 cache), the page offset provides enough information to infer the location in the cache, as demonstrated in [2]. However, more sets the cache has, the more bits from the $pfn$ are needed to specify a location.
- **LLC slices**: In order handle several concurrent LLC accesses, Intel processors usually divide their LLC in slices, with an unpublished slice selection algorithm distributing the memory blocks among them. This means that even if we can calculate the cache set of our data, we still need to locate it in one of the slices.

Indeed all of these complications can be handled to mount a LLC *Prime+Probe* attack, as demonstrated in [21], [22]. The first issue can be solved by monitoring only the target sets where the targeted memory block location is known. The second issue can be solved by using hugepages as described in II-B. Hugepages are usually in the order of MBs, making the $p_o$ larger than the usual 12 bits that is obtained with regular pages. With 2MB pages, 21 bits of the page offset is visible and sufficiently large to target modern LLCs. The final problem can be solved if the slice selection algorithm is known or by creating a large pool of memory blocks and detecting which memory blocks collide in the same slice, as demonstrated in [30], [31], [32].

For CPUs with non-linear slice selection algorithms, it is harder to recover the T-table location in cache. For instance, with a 10 core machine with 20 MBs of cache, even if we know the last 6 bits of the set number of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



T-table there are still 5 unknown bits. In addition, the T-table can be located in non-linearly or linearly addressed slices of the cache. Hence, there are total $2^5 \times 10$ possible set-slice pairs for T-table location. To find the correct set-slice pair, all 320 possibilities must be profiled before an actual *Prime+Probe* side channel attack.

## III. The AES Attack

We study two different side-channel attacks known as *Flush+Reload* and *Prime+Probe* to monitor accesses to memory blocks. Both techniques can be implemented in the cross-VM setting, but while the *Flush+Reload* technique requires de-duplication features to be enabled by the hypervisor, *Prime+Probe* does not require specific characteristics to succeed. This de-duplication process is only applied if data is marked as shared (as is the common case for all crypto libraries).

### A. A single cache line attack on AES

In this section we explain how the AES encryption is implemented in the T-table C reference implementation of `OpenSSL 1.0.1g` library and how the leakage is exploited. AES usually performs 4 operations per round, i.e., `SubBytes, ShiftRows, MixColumns and AddRoundKey`. T-table implementations usually mix the `SubBytes, ShiftRows and MixColumns` operations in a single T-table look-up operation and a XOR operation. Thus, T-table AES implementations are only based on Table look-up and XOR operations. Since the last round does not perform the `MixColumns` operation, the last round key byte is directly related with the cyphertext byte and the T-table access through a simple XOR addition:

$$C_i = T_j[S_i] \oplus K_i^{10} \tag{1}$$

where $T_j$ is the corresponding T-table applied to the $i$th byte and $K_i^{10}$ ($i$th byte of the last round key). In this work, we exploit the fact that if both the ciphertext and the accessed T-table position are known by the attacker, a simple XOR will output the key byte that was used in the last round. More details on the T-table AES implementation can be found in [22].

`OpenSSL 1.0.1g` utilizes four 1KB size T-tables. Due to the facilities that the last round gives us, we focus our attack in the last round of AES. Recall that this is not an issue, since the key scheduling is invertible. Therefore, our goal is to determine the accessed T-table lines used in the last round of an AES encryption. In addition to the accessed table positions, we also assume that the attacker has access to the corresponding ciphertext $c$. Therefore, it is assumed that the attacker has several tuples $\langle c, t \rangle$.

These accessed T-table positions will be known with the help of two spy processes: *Flush+Reload* and *Prime+Probe*. However, a cache line will contain more than one table positions, meaning that we cant recognize an access to a single table position but rather to a entire memory block. With 64 byte memory lines, each T-table occupies 16 cache lines and each cache line holds 16 T-table positions for `OpenSSL 1.0.1g`. Furthermore the sets that each of these lines occupy in the cache increase sequentially, i.e, if $T[0-15]$ occupies set 0, then $T[16-31]$ occupies set 1..etc. Since each encryption makes 40 accesses to each of the T-tables, the probability of not accessing one of the T-tables memory lines is:

$$\text{Prob}[\text{no access} T[i]] = (1 - (n/256))^l \tag{2}$$

For AES-128 in `OpenSSL 1.0.1g`, $n = 16$ and $l = 40$ per $T_j$, therefore $100\% - \epsilon_0$ of reloads are expected to come from the cache in $H_0$, and only $92\% + \epsilon_1$ for $H_1$, where $\epsilon_i$ are noise terms. Hence, a side-channel containing information about memory/cache accesses will feature differing leakage distributions $f_0$ and $f_1$ for cases $H_0$ and $H_1$, respectively. *Flush+Reload* and *Prime+Probe* techniques distinguish these distributions.

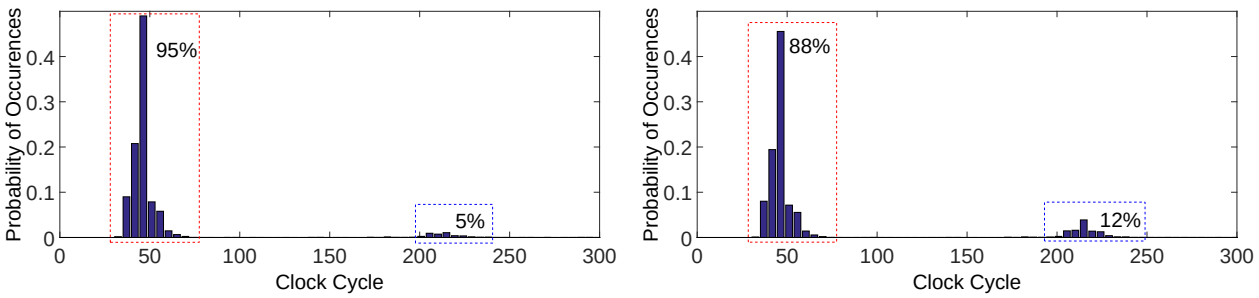
### B. Flush+Reload and Prime+Probe applied to AES

In order to recover the AES key, we apply the two spy processes presented in section II. They both have similarities and differences that we will discuss below:

- ***Flush+Reload*:** The first step of the attack is flushing two targeted memory blocks related to the 4 T-tables before the encryption starts to ensure that they are located in the memory. Then, the attacker waits until the encryption is completed and checks whether targeted memory blocks (i.e. a position in the T-table) have been used or not. The *Flush+Reload* side channel attacks offered a high distinguishable covert channel due to significantly different distributions, as it can be seen in Figure 4. On the other hand, the measurements include noise from various sources. One of which is measurement inaccuracy stemming from micro architecture, by the OS and the hypervisor. In general, this noise causes a moderate increase in the number of cycles. However, if e.g. a context switch happens during a measurement, the value might be off several orders of magnitude. In order to remove this noise, a threshold can be applied to filter out the outliers.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems






(a) Distribution $f_0$ for case $H_0$ (b) Distribution $f_1$ for case $H_1$

Fig. 4. Leakage Distributions $f_0$ and $f_1$ if Hypotheses $H_0$ and $H_1$ are correct. The measurements were taken in an Intel i5 2430M CPU with the *Flush+Reload* attack [37].

Having said that, even with a reasonable threshold, the noise is definitely not Gaussian, possibly better described as Ex-Gaussian. The second source of noise is the cache misses due to cache line evictions by another process and is independent of the measurement process.

– ***Prime+Probe*:** The *Prime+Probe* attack can be similarly applied to obtain the AES key. However, since we no longer share the T-table memory blocks with the victim, we first have to determine which set in the LLC the T-tables go into. If all T-tables reside in one memory page, it is sufficient to know the location of one since the rest will go to the adjacent sets. In order to trigger this information, we feed the encryption server with random plaintexts. Since we know that the probability of not using a specific memory block is 8% and with sufficient number of encryption we should see that distribution in the correct set. Once the correct sets are found, we just need to prime before the encryption to make sure that the monitored memory block will reside in the memory. After the encryption, we probe our memory lines to check if the set was used. The distribution of an accessed T-table position measured with *Prime+Probe* can be seen in Figure 5.
Although the noise sources are very similar in both attacks, *Prime+Probe*is more easily affected by them. In fact, with *Flush+Reload* a noisy process needs to create $w$ memory lines pointing to the set where our monitored memory block resides to create an undesired eviction, being $w$ the associativity. However, with the *Prime+Probe* technique, a single memory block pointing to the monitored set by a noisy process will already create an eviction of our primed memory blocks.

### C. Searching AES T-table Locations

AES implementation of `OpenSSL 1.0.1g` uses static T-table entries i.e. all T-tables are always in the same position in the LLC for every encryption after compiling the library. Therefore, we need to find the T-table locations in the LLC to perform the attack.

In the Intel Xeon E5-2670 v2 processors, the LLC has 10 slices and each slice has 2048 sets in total. Also, a non-linear slice selection algorithm is implemented and reverse engineered in [24]. The lower 12 bits of the physical addresses of T-tables are known by the attacker which helps to decrease the possible locations for T-tables in the LLC. Therefore, we need to profile every set that solves s mod 64 = o, where s is the set number and o is the offset for a T-table address. The total number of possibilities per T-table is 320 since there are 32 possible set numbers and 10 different slices. The way to detect the correct set-slice pair is to monitor all 320 candidates.

If the machine has $2^n$ cores it has linear selection algorithm where it is easier to find the T-table positions in the LLC because we only have to create n eviction sets (the lower 16 bits do not play a role in the slice selection). Thus, one can accordingly change the lower 16 bits to select the set he wants to target, since the slice will not be affected). However, if the non-linear slice selection algorithm is implemented in the machine, the lower 16 bits of the address are taken into account (non-linearly) to select the slice. Therefore, we need to implement the Algorithm-1 to create 320 eviction sets easily to increase the speed of the attack (rather than creating all eviction sets one by one).

Therefore we propose an algorithm 1 to find all the lines for every set-slice pair. By using this algorithm, we first discover which lines belong to which linear slice in the LLC, then create all 320 eviction sets automatically. This algorithm can also be used to profile the whole LLC

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



in the cloud as long as a processor with similar cache architecture is present.

**Algorithm 1** T-table Searching Algorithm

```
Input: Set number (S), Linear slice (S_x)
Assume working with the linear slices
for i from 0 to 255 do
    if Line_i belongs to S_x then
        if nl(Line_i)==0 then
            Add Line_i to List(L_l)
        else
            Add Line_i to nonlinear list(L_nl)
        end if
    end if
end for
```

### D. Selecting the Outliers Cutoff Threshold

In the experiments, we have used different thresholds for outliers i.e. cutoff points where we treat any data surpassing it as noise. In cross-VM attack scenario, since the attacker has legitimate access to the same physical machine as the victim, he can simply run the encryption himself, obtaining the necessary execution and memory/cache access times. Therefore the cutoff threshold selection can be done in real world scenarios.

## IV. Distinguishers for the AES Attack

To analyze the side-channel data, we describe and compare three different distinguishers. The aim is to process one byte of the ciphertext *c* with the corresponding T-table entry and the access time *t* to recover the one byte *k* of the last round key.

As mentioned earlier, we have two different sets of hypothesis to observe the leakage. If the hypothesis is correct, the distributions $f_0$ and $f_1$ for two sets differ and hence they become distinguishable with sufficient number of samples. In contrast if the hypothesis is wrong, both distributions have the data from the mixed distribution and this makes them indistinguishable. Therefore, we apply different distinguishers to decide whether samples for hypotheses $H_0$ and $H_1$ are actually from different distributions.

In the field of side-channel attacks, the most commonly used distinguisher is the difference of means of two distributions [33], [34]. As for the zero-value DPA [35], our hypothesis deviates from a single-bit prediction, yet the test still distinguishes two cases. Similarly the variance test uses a statistical moment to distinguish the two distributions [36], [34], [33]. The last distinguisher applies a *miss counter*, as used in [16]. The list is neither exhaustive nor do we make an optimality claim.

*Miss-counter based Distinguisher:* This distinguisher counts and compares the memory misses for the two cases $H_0$ and $H_1$. Ideally, there should be no misses for $H_0$, as the memory block must have been accessed by the AES execution. To establish a *miss counter*, reload timings are converted to hits (0) or misses (1) with respect to the threshold value. Since $H_1$ contains significantly more values than $H_0$, we compare the relative counters instead of absolute ones. Finally, our distinguisher becomes:

$$\mathcal{D}_{miss\_ctr} = \arg\max_{\hat{k}} \left(\overline{ctr}_{H_1} - \overline{ctr}_{H_0}\right)$$

*Difference of Means Distinguisher:* The difference of means distinguisher approximates the means of two distributions and outputs their difference in cycles.

$$\mathcal{D}_{means} = \arg\max_{\hat{k}} \left(\overline{\tau}_{H_1} - \overline{\tau}_{H_0}\right)$$

Since $H_0$ should feature more cache accesses than $H_1$, $\overline{\tau}_{H_0}$ is expected to be smaller, i.e. the largest positive difference corresponds to the most likely key hypothesis. Welch's t-test distinguisher (which divides the means with their respective variance) can be equally well applied to guess the correct key. Indeed, Welch's t-test is commonly applied to check two hypothesis where two gaussian distributions have different means and variances. In this work, we studied Welch's t-test and did not obtain an improvement over the difference of means distinguisher. Thus, we use the difference of means distinguisher due to its simplicity.

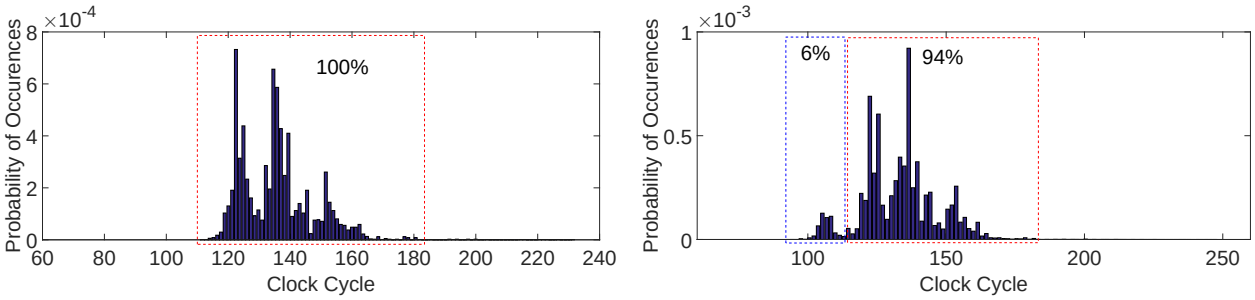
*Variance based Distinguisher:* This distinguisher outputs the difference of variances in cycles.

$$\mathcal{D}_{vars} = \arg\max_{\hat{k}} \left(var(\tau_{H_1}) - var(\tau_{H_0})\right)$$

Note that again the variance of $H_0$ should be smaller than that of $H_1$. However, outliers can badly affect this distinguisher. In cache attacks, significant outliers can be orders of magnitude larger than the regular data and need to be filtered. Since $H_i$ is key dependent, the guessed key $\hat{k}$ that maximizes the difference is most likely key candidate. Note that the sign carries information in all three tests. In fact, the case $H_0$ and its leakage $f_0$ correspond to fewer cache misses hence lower miss counter, lower average (mean) access time and lower variance. Our results confirm that taking the sign into

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems






(a) Distribution $f_0$ for case $H_0$

(b) Distribution $f_1$ for case $H_1$

Fig. 5. Leakage Distributions $f_0$ and $f_1$ if Hypotheses $H_0$ and $H_1$ are correct. The measurements were taken on a 10 core Intel Xeon E5-2670-v2 CPU with the *Prime+Probe* attack.

account derives a better distinguisher. Details of the distinguishers used in the attack can be found in [37].

When we compare three distinguishers, we observe that the miss counter is the most useful for this study. It is quite intuitive, as cache misses and hits are what we are looking for. Furthermore, the method is only marginally affected by outliers. The main disadvantage of this method is the requirement of a threshold, which is processor-dependent and requires some minimal profiling. The other two methods are more affected by outliers. All three distinguishers can easily be converted to a correlation method. Indeed, by correlating the right term (e.g. $\overline{\tau}_{H_0}$) to 0 for $H_0$ (a guaranteed cache hit with low reload time) and 1 for $H_1$ (a possible cache miss with higher reload time), the most likely key $\hat{k}$ features the highest correlation.

## V. Experiment Setup and Results

### A. Lab Experiment Setup

We use two setups to quantify the additional noise stemming from virtualization;

- **Native Execution:** In this setup, both the AES encryption process and the attacker run on a native Ubuntu 12.04 LTS version with no virtualization. In this setting, we used a 2-core Intel i5-2430M CPU clocked at 2.4 GHz. By running the attack in this setup, we minimize the environmental noise and achieve comparability to former non cross-VM cache attacks.
- **Cross-VM Execution:** In this setup, two up-to-date Ubuntu VMs are launched and managed by VMware ESXI 5.5 bare-metal hypervisor. The attacks are then performed in Cross-VM setting, overcoming hypervisor isolation boundaries. The first VM is used as the target that performs the AES encryption while the second VM acts as the attacker trying to recover the secret key. The experiments in this setting were performed on a 10 core Intel Xeon E5-2670-v2 CPU. This setup reflects a realistic attack scenario by using a modern CPU used in commercial clouds [38], [39]. In this setup, accesses from the cache and the memory take around 30 and 233 cycles respectively. In the same setup, single AES encryption operation take 659 and 257 cycles for with and without pre-flushed T-tables.

Note that all the timing measurements in the experiments are gathered using the *Read Time Stamp Counter and Processor ID (RDTSCP)* instruction, which not only reads the time stamp counter but also implementes serializing instructions to ensure that all memory operations have finished before we start reading. The usage of the RDTSCP instruction is allowed in VMware user mode, but not in KVM. In this case, the *Read Time Stamp Counter (RDTSC)* with a *mfence* instruction can be used. Moreover, these instructions are not emulated by the hypervisor but executed directly, unlike other serializing instructions like the *CPUID* used in [16]. Also, the flushing operation is performed using the *Cache Line Flush (CLFLUSH)* instruction.

### B. Flush+Reload Results

These results are obtained for two environment in both native and virtualized environments as presented in [37]. In addition, the timing behavior is analyzed to show the improvement on the success rate by using the three different distinguishers mentioned in Section IV: the miss counter, the difference of means and the difference of variances.

**Native results:**

At first we present and compare the scores of the key guesses using the three different distinguishers in native execution in Figure 6. The difference of means

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



and variances distinguishers suffer more from noise due to heavy outliers stemming from different microarchitectural sources of noise. However the measurements shown in Figure 6 were taken by cutting off outliers with a threshold value of 5 times the memory access time. It can be seen that for 10,000 encryptions the three distinguishers maximize the score for the correct key, 180 in this case.

Finally the number of traces needed for the recovery of the key are presented in Figure 7.

**Cross-VM Execution Results:**

In the cross-VM setting, the attack requires 30,000 encryptions to recover the full key using the miss counter hypothesis as seen in Figure 8(a). In the same setting, 50,000 encryptions are needed when the difference of means distinguisher is used, shown in Figure 8(b). Finally, we would like to remark that **only 15 seconds** are enough to recover the whole key, which to the best of our knowledge is the fastest working attack in a realistic cross-VM setting without scheduler exploitation.

### C. *Prime+Probe* Results

One of our improvements with respect to [22] is reducing the noise and increasing the success rate. The results are summarized as follow:

**Native Results:**

In order to cope with the outliers, which affect mean and variance based distinguishers significantly, we implemented an upper threshold of 500 cycles. The success of the distinguishers can be seen in Figure 9. All distinguishers are able to distinguish the hypotheses $H_0$ and $H_1$ easily.

For this attack scenario, we show that 10,000 encryptions are enough to find the correct key byte using the mean distinguisher (Figure 10(a)). The mean distinguisher works better with the *Prime+Probe* attack because it is more prone to noise than the *Flush+Reload* attack. Therefore we use it to calculate the number of encryptions needed to recover the full AES key in Figure 10(a). As pointed out earlier, the advantage of using mean based distinguishers is that it does not require a threshold to distinguish between cache accesses and memory accesses.

**Cross-VM Execution Results:**

In order to simulate the cross-VM scenario we are reproduce the *Prime+Probe* attack across co-located VMs using VMware as our hypervisor In this setting, finding the T-table locations in the cache becomes difficult due to the tremendous noise. Therefore, we need more samples and repeated experiments to find the locations of T-table entries. After we find the T-table entries in the cache the *Prime+Probe* attack is performed. Again due to the additional layer of noise added by the hypervisor, more encryptions traces are needed compared to the native case. To extract the full key we need at least 40,000 ciphertext-timing tuples. After the experiment we apply the distinguishers to find out which one works better. The results can be seen in the Figure 11. For the cross-VM scenario the difference of variances is not successful due to the observed noise in the square operation. Even if we eliminate the outliers noise causes the higher variance change due to the overwhelmed noise in the square operation. The miss counter and difference of means distinguishers work similarly well. Hence, both of them can be applied to recover the correct key in the cross-VM scenario. However, the difference of means distinguisher can adapt itself better than the miss counter distinguisher. Therefore we use it in the results presented in Figure 10(b). Note that only after 40,000 encryption the correct key is distinguishable easily, meaning that cross-VM attack needs 4 times as many number of encryptions than in the native case. Note that we require less number of encryptions than in [22] even with the same distinguisher due to a better handling of undesired noise. Further note we use more sophisticated machines with a higher number of cores than those in [22] and therefore the noise spreads more easily over them.

### D. *EC2 Public cloud experiment setup*

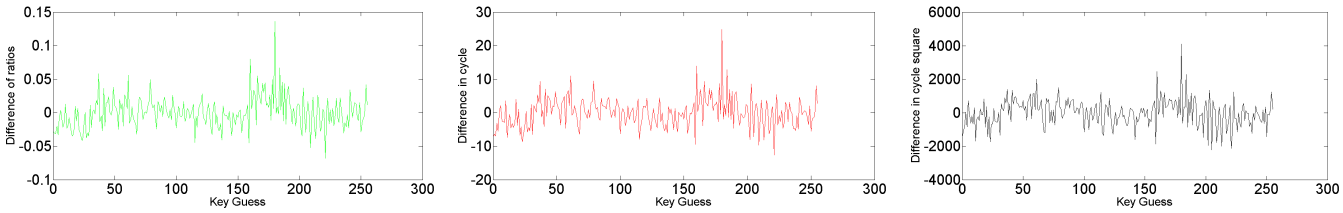
After successfully applying both the *Flush+Reload* and *Prime+Probe* attacks in our controlled lab setup, we verify if they are applicable in a realistic cloud. We chose Amazon EC2 due to its high adoption in the market. Amazon EC2 uses a modified version of the Xen hypervisor, which does not utilize de-duplication. This fact makes the *Flush+Reload* attack infeasible in their system. However, guest OS can still allocate huge size pages in Amazon EC2. Therefore, the *Prime+Probe* side channel attack is applicable in EC2 instances.

We first need to have two co-located VMs to successfully run the attack. In order to do so, we utilize the same technique used in [24], i.e., we create cache contention in a specific set in two VMs and check whether we observe the contention across co-located VMs. We utilize medium type instances, which use the same Intel Xeon 2670-v2 that we used in our lab setup.

Once the co-location is achieved, we replicate our lab scenario, i.e., a virtual machine runs an AES server while the co-located VM requests encryptions as the *Prime+Probe* spy process executes.

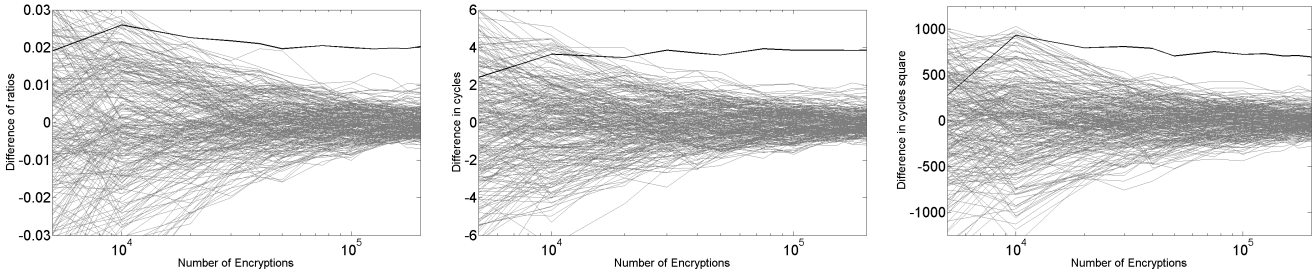
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems





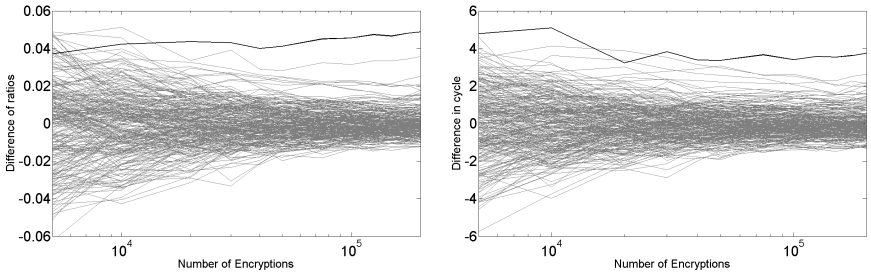

(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 6. Comparison of the key guess scores in the natively executed scenario using *Flush+Reload* for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 10,000 traces. The correct key is 180 and clearly distinguishable in all three cases [37].




(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 7. Comparison of results with varying traces in native execution using *Flush+Reload* for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c) [37].




(a) Miss Counter (b) Difference of Means

Fig. 8. Results in cross-VM execution utilizing *Flush+Reload* and the miss counter distinguisher (a) and the means distinguisher (b) [37].

### *E. Amazon (EC2) Results*

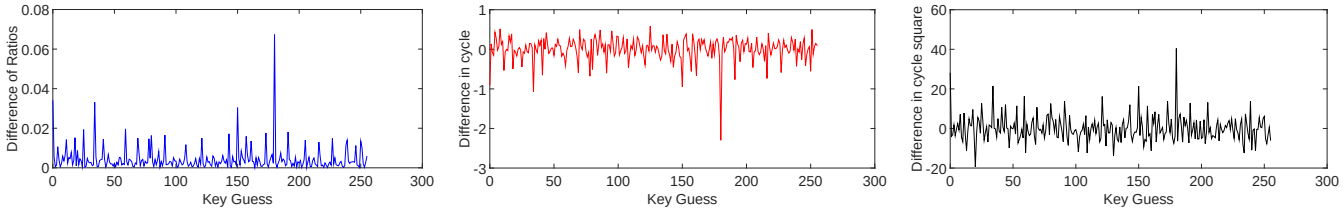
In this setting, again the T-table location is difficult to obtain. In consequence, we need to repeat the T-table searching algorithm several times to find out the correct set-slice pair. After finding the T-table location we implemented our experiment with 400,000 encryptions. The experiments take more time to complete due to the slow speed of transmission of ciphertexts and plaintexts between VMs. After obtaining all ciphertext-timing pairs the three distinguishers are applied to recover the correct key. The results can be seen in Figure 12. Figure 13 shows that the required number of encryptions is higher than other scenarios and at least 200,000 encryptions are needed to recover the correct key.

### *F. Discussion of Results*

Table I summarizes our results. We clearly observe that the number of encryptions that *Flush+Reload* requires is smaller than *Prime+Probe*, i.e., as low as 30,000 traces to successfully recover the key. Since *Prime+Probe* monitors an entire set, it is less resilient by undesired LLC accesses, and more traces are needed to recover the key. An interesting fact that we observed is that the miss counter distinguisher needs the same number of encryptions to successfully recover the key in both our lab environment and the EC2 public cloud. We needed 75,000 encryptions (less than a minute) to recover a 128 bit AES key in a public cloud, demonstrating that cache side-channel attacks are a threat

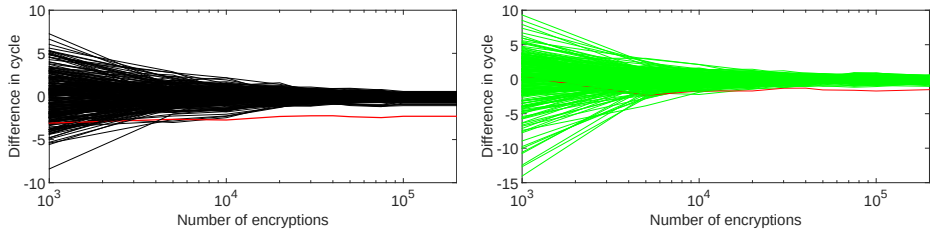
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems





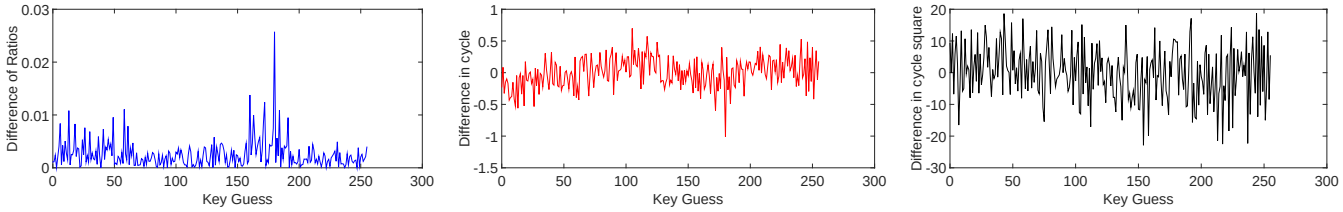

(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 9. Comparison of the scores of key guesses in the natively executed scenario using *Prime+Probe* for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 200,000 traces. The correct key is 180 and clearly distinguishable in all three cases.




(a) Ctr Dist. (b) Mean Dist.

Fig. 10. Number of encryptions required to recover the correct AES key using the *Prime+Probe* attack with mean based distinguishers in (a), Native scenario (b), cross-VM scenario




(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 11. Comparison of the scores of key guesses in the cross-VM scenario using *Prime+Probe* for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 200,000 traces. The correct key is 180 and clearly distinguishable for miss counter and difference of means distinguishers.

across co-resident tenants. In applications such as Netflix and Dropbox where large amounts of encrypted data is transferred over the network, the attacker has enough samples to recover a session key.

## VI. Countermeasures

### A. Hardware Based Countermeasures

There are several approaches that can be taken to avoid cache interference between co-located users. In fact, Wang et al. [40] proposed two countermeasures to avoid cache leakage detection. We discuss their applicability to stop the attacks that have been described in this paper:

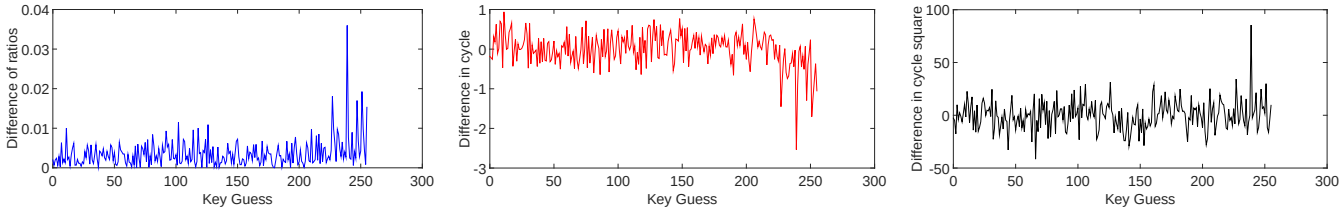
– **Dynamic Cache Partitioning**: This countermeasure blocks a cache line being used by an application so that it cannot be evicted upon a cache miss. The countermeasure is not effective against the *Flush+Reload* attack, specially if the ”locked” bit can be changed by the attacker. If the attacker manages to unlock the bit in the memory block

TABLE I
Comparison of the required encryption number for different attack scenarios and platforms

| | Distinguishers | | |
|---|---|---|---|
| Nonlinear Machine | Miss counter | Mean based | Variance based |
| *Flush+Reload* on VMware | 30,000 | 50,000 | 75,000 |
| *Prime+Probe* on VMware | 75,000 | 40,000 | 200,000 |
| *Prime+Probe* on EC2 | 75,000 | 100,000 | 100,000 |

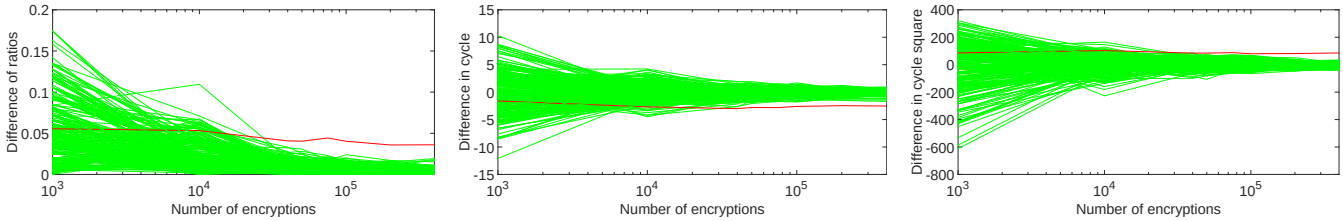
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems






(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 12. Comparison of the scores of key guesses in EC2 using *Prime+Probe* executed for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 400,000 traces. The correct key is 239 and clearly distinguishable for all distinguishers.




(a) Ctr Dist. (b) Mean Dist. (c) Var. Dist.

Fig. 13. Comparison of the scores of key guesses in EC2 using *Prime+Probe* for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to different number of traces.

that *shares* with the victim, the countermeasure is not effective. Furthermore, the *clflush* is a powerful instruction to ensure cache coherence for those systems that lack of this ability. Thus, even if the attacker does not have the right to unlock the cache line, it is an open question whether such an instruction is powerful enough to ignore the locking bit. As for the *Prime+Probe* attack, the countermeasure would be effective since the attacker would not have the ability to fill a specific set if any other process has locked lines on it.

– **Random Permutation Cache**: The basic idea behind this countermeasure is to choose a random cache line in a random set when an eviction needs to be performed. Again, this countermeasure would not prevent the *Flush+Reload* attack, since the attack does not base its procedure on cache line eviction but on accesses to a shared memory block. Thus, the *clflush* command should still be able to flush the shared memory block from the cache, and the attacker should still be able to access the same memory block in the cache. However, the *Prime+Probe* attack would be prevented, since upon an eviction request made by the victim the primed set would likely not be evicted due to the implemented randomness.
– **Hardware cryptographic primitives**: Hardware cryptographic primitives: As for cryptography, the memory based leakage (and thus, the cache attacks) can be avoided if users utilize hardware implementations provided by CPU vendors. In fact, if all the cryptographic operations are performed in hardware, the CPU cache is not utilized at all and the cache attacks are no longer possible. This is the case for AES in Intel processors, which support Intel AES-NI. For the purposes of this study, we consider these hardware implementations to be secure. Nevertheless, these types of attacks might be able to extract leakage from other cryptographic (or non-cryptographic) applications such as RSA (as it was shown in [15], [24]) for which a hardware implementation is not provided.

### B. Software Based Countermeasures

Software countermeasures mainly imply a constant flow execution, where the attacker cannot distinguish between accessed and non-accessed patterns. In order to achieve this behavior, `OpenSSL` offers a cache leakage free implementation, where all memory tables are loaded in the cache prior to the first and last rounds. This avoids the attack presented in this work, since the flushing/priming resolution is not precise enough to perform the attack during the last round.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2016.2550438, IEEE Transactions on Multi-Scale Computing Systems



## VII. Conclusion

In conclusion, we showed that there is a very real threat in using insecure software implementations of AES on virtualized systems without AES-NI hardware support. We demonstrated two different attacks namely *Flush+Reload* and *Prime+Probe*. In all these attack scenarios, we succeed in recovering the full AES secret key using varying number of samples.

## References


[1] M. Neve and J.-P. Seifert, "Advances on Access-Driven Cache Attacks on AES," in *SAC*, 2007, pp. 147–162.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," ser. CT-RSA'06.

[3] W.-M. Hu, "Lattice Scheduling and Covert Channels," in *Proceedings of the 1992 IEEE Symposium on Security and Privacy*.

[4] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *J. Comput. Secur.*, vol. 8, no. 2,3, pp. 141–158, 2000.

[5] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," 2002.

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

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

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

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

[10] O. Acıiçmez, "Yet Another MicroArchitectural Attack: Exploiting I-Cache," in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.

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

[12] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis," in *IEEE: Security & Privacy*, 2011.

[13] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS 2012*, 2012, pp. 305–316.

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

[15] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *(USENIX Security 14)*, pp. 719–732.

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

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *CCS*, 2014, pp. 990–1003.

[18] G. Irazoqui, M. S. İnci, T. Eisenbarth, and B. Sunar, "Lucky 13 Strikes Back," ser. ASIA CCS '15, 2015, pp. 85–96.

[19] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way." in *CHES*, 2014, pp. 75–92.

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

[21] Liu, Fangfei and Yarom, Yuval and Ge, Qian and Heiser, Gernot and Lee, Ruby B, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015, pp. 605–622.

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing?and its application to AES," *IEEE S&P*, 2015.

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

[24] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud," IACR Cryptology ePrint Archive, Tech. Rep., 2015.

[25] "Kernel Samepage Merging," 2015, http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb.

[26] M. T. Jones, "Anatomy of linux kernel shared memory," 2010, http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/.

[27] "Kernel Samepage Merging," 2015, http://www.linux-kvm.org/page/KSM#Kernel_Samepage_Merging.

[28] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest OS," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 1.

[29] Suzaki, Kuniyasu and Iijima, Kengo and Yagi, Toshiki and Artho, Cyrille, "Effects of memory randomization, sanitization and page cache on memory deduplication," 2012.

[30] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors," in *Euromicro DSD*, 2015.

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

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

[33] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *COSADE*, 2015, pp. 111–126.

[34] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A Practical Implementation of the Timing Attack," in *Smart Card Research and Applications*, pp. 167–182.

[35] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO 99*, pp. 388–397.

[36] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer, 2008, vol. 31.

[37] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO '96*, pp. 104–113.

[38] "Amazon EC2 Instances," http://aws.amazon.com/ec2/instance-types/.

[39] "Google Compute Engine Instance Types," https://cloud.google.com/compute/docs/machine-types.

[40] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.