A Faster and More Realistic *Flush+Reload* Attack on AES

Berk Gülmezoğlu, Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, Berk Sunar

Worcester Polytechnic Institute, Worcester, MA, USA
{bgulmezoglu,msinci,girazoki,teisenbarth,sunar}@wpi.edu

Abstract. Cloud's unrivaled cost effectiveness and on the fly operation versatility is attractive to enterprise and personal users. However, the cloud inherits a dangerous behavior from virtualization systems that poses a serious security risk: resource sharing. This work exploits a shared resource optimization technique called memory deduplication to mount a powerful known-ciphertext only cache side-channel attack on a popular `OpenSSL` implementation of AES. In contrast to the other cross-VM cache attacks, our attack does not require synchronization with the target server and is *fully asynchronous*, working in a *more realistic* scenario with much weaker assumption. Also, our attack succeeds *in just 15 seconds* working across cores in the cross-VM setting. Our results show that there is strong information leakage through cache in virtualized systems and the memory deduplication should be approached with caution.

Keywords: Asynchronouos Cross-VM Attack, Memory Deduplication, Flush and Reload, Known Ciphertext Attack, Cache Attacks

1 Introduction

Cloud computing and virtualization is popular more than ever with large companies like Microsoft, Google, Amazon, IBM, Oracle, Rackspace and many others investing billions of dollars trying to get a foothold in this new area of lucrative business. This rapid increase in the number of cloud service providers is directly related to the emergence of server-less companies like Netflix, Dropbox, Instagram, Pinterest, Reddit, Imgur and many others that are using commercial cloud infrastructure [10]. Instead of buying expensive servers without knowing exactly how many of them they need, and then hiring IT personnel to maintain those servers, these fast growing companies have chosen to use public cloud systems to maintain their software and services.

The opportunities of using the commercial cloud are fairly obvious however, threats are not. Sharing a physical system between users reduces the cost while increasing the utilization hence the productivity. The isolation between the Virtual Machines (VM) in these systems is maintained by the Virtual Machine Manager (VMM) at the software level. However, software layer confinement techniques that force the *sandboxing* does not guarantee complete isolation and cannot ensure the prevention of data leakage from one VM to the other. The most common source of information leakage across VM boundaries is the shared cache and the memory of the underlying physical system. Particularly memory deduplication allowed researchers to mount attacks that threaten both the user privacy and the security of the cryptographic systems.

In 2009, Ristenpart et al. [24] showed that it is possible to co-locate with a target on a cloud environment, namely Amazon EC2, and extract keystrokes from the co-located VM. In 2011, by exploiting the Kernel Samepage Merging (KSM), Suzaki et al. [25] was able to detect processes like `sshd`, `apache2`, `IE6` and `Firefox` running on a co-resident VM. The significance of this study is that it is possible to not just detect the existence of a target VM, but also detect running processes.

Recently in 2013 Yarom et al. [29] applied the *Flush+Reload* attack across VMware VMs to recover a RSA key. Later in 2014, Irazoqui et al. [14] used Bernstein's AES cache timing attack to partially recover an AES key from various AES crypto library implementations in a cross-VM setting under XEN and VMware ESXI hypervisors. Also in 2014, Irazoqui et al. [15] implemented a cross-VM access driven cache attack on AES in a VMware ESXI system using the *Flush+Reload* attack.

Our Contribution

In this work, we implement for the first time a known-ciphertext cross-VM attack on AES using the *Flush+Reload* method and use three distinct data analysis methods to fully recover the secret key with varying encryption observations for different scenarios. For the attack, we take advantage of VMware ESXI' s memory deduplication mechanism called the Transparent Page Sharing. The attack is mounted on a multi-core high-end server, a specification found commonly on commercial cloud systems and does not require the attacker and the victim to be running on the same physical CPU core. Compared to the attack in [12], our attack is minimally invasive and works with less assumptions since the attacker does not need to control or exploit in any way the target process execution. Also compared to [15], the new attack does not assume to have access to the encryption server and works only by listening to the encryption server via cache covert channel and obtaining the ciphertexts from the network channel.
In summary, this work

- for the first time, mounts a cross-VM, **known-ciphertext only** AES key recovery attack using the *Flush+Reload* technique
- improves upon the previous cross-VM AES cache attacks by flushing **in between** the encryption rounds
- presents three distinct analysis methods that can be adapted to any table-based block ciphers

2 Cache Side-channel attacks

Cache side-channel attacks exploit microarchitectural leakages stemming from memory access time variations, e.g. when the data is retrieved from small faster memories called caches as opposed to slow bulk memories. Caches are useful due to two main principles, i.e. the *temporal* and *spatial locality. Temporal locality* predicts that recently accessed memory locations are likely to be accessed soon again, while *spatial locality* predicts that data located nearby the accessed memory locations are also likely to be accessed soon. In general, caches hold not only recently accessed data, but also an entire cache line containing data in nearby locations. In modern CPUs, caches are organized into multiple levels L1, L2 and L3 where the first two levels are smaller and core exclusive, while the last level is considerably larger and is shared across cores. While retrieving data from any level of the cache is faster than retrieving the same data from the main memory, higher levels of the cache are even faster than lower levels. L1 being the fastest, L3 the slowest, different cache levels have different access times which enable attacks like *Prime+Probe* to distinguish between accesses to the L1 cache and to the L3 cache. In addition, the last level of cache is shared between all cores, giving the attackers the opportunity to use it as a covert channel between cores and mount attacks such as *Flush+Reload*.

2.1 Related Work

The first theoretical consideration on the extraction of information via cache memories was demonstrated by Hu [13], whereas 6 years later Kelsey et al. [18] expanded this consideration by suggesting the presence of cache leakage due to the hit/miss ratio. Following up Kelsey's work, Page described a theoretical chosen plaintext attack based on the collection of cache profiles [23]. One year later Tsunoo et al. [26] proposed the first practical implementation of cache attacks on the DES cryptographic algorithm.

The first practical cache side-channel attack on AES appeared in 2005 by Bernstein [7] showing that the table look up operations from different cache lines have different access times in an AES encryption. Further, he showed that using this cache access time information, an adversary can recover secret encryption key from a popular AES implementation, i.e. the `OpenSSL` cryptographic library. In a similar attack, Osvik et al. [22] presented two spy processes that are able to monitor the cache usage: `evict+prime` and `prime+probe`. Although the latter one proved to be significantly more efficient, both spy processes recovered the AES encryption key used by an `OpenSSL` server. A few months later, Bonneau and Mironov [8] and Acıiçmez and Koç [5] presented new attacks targeting AES that exploited internal table look up collisions in the cache during the last and first rounds respectively. In spite of the prominent successful attacks on symmetric key cryptography, public key cryptography was also considered a popular target for cache side-channel attacks. Indeed in 2007, Acıiçmez demonstrated the usage of the `prime and probe` spy process in the instruction cache against a RSA encryption.

Cloud computing systems became the next challenge for side-channel attack researchers. In 2009, after Ristenpart et al. [24] demonstrated that they were able to co-locate an attacker's virtual machine (VM) with a potential victim's VM with a success probability of 40% in the Amazon EC2 cloud. Even further, the authors managed to recover keystrokes from the co-resident victim's VM, showing that the cache side-channel attacks are both practical and applicable to real world scenarios. The possibility of co-location fueled further research on new cache side-channel techniques and cache leakages in VMs. For instance, in 2011 Chen et al. improved over the previous RSA attacks in the instruction cache [9] while Gullasch et al. discovered a new side-channel technique that would later be called the *Flush+Reload* [12]. The *Flush+Reload* attack recovered AES secret keys by taking control of the Completely Fair Scheduler (CFS) [16, 3]. At the same time, previous side-channel techniques such as *Prime+Probe* were also adapted to work in virtualized settings by Zhang et al. [30, 31]. They utilized a spy process to detect co-resident tenants and to recover El Gamal encryptions keys. More recently, Yarom and Falkner [29] applied the *Flush+Reload* technique to recover, for the first time, RSA encryption keys across VMware and KVM VMs. Shortly later Benger et al. [6] demonstrated the viability of the *Flush+Reload* technique to recover ECDSA encryption keys. Finally, Irazoqui et al. [14, 15] recovered AES keys in virtualized environments with Bernstein's attack and the *Flush+Reload* technique.

2.2 Memory deduplication

Memory deduplication is an OS memory optimization technique that allows the OS to keep only a single copy of a data in the memory when multiple processes are using the same data. While this feature is useful in native execution, it is even more useful in virtualized setting where many VMs use the same OS and/or the same software.

Hence, to reap the benefits of the deduplication, VMMs have also implemented memory deduplication techniques to allow more VMs to run on the same physical machine. For this, the VMM recognizes identical and redundant memory copies by first checking their hash values and then performs a bit-by-bit comparison. If the memory content is determined to be shared by more than one process/VM the memory manager removes multiple copies from the memory. Note that even though this deduplication process is only performed on shareable memory pages like shared libraries, shared libraries are used in many software packages. Memory deduplication methods are especially effective when hosting multiple processes, as is the case in virtualized systems. Consequently, VMMs like VMware [27, 28] and KVM [4, 17] implement variations of memory deduplication, i.e. Transparent Page Sharing(TPS) and Kernel Samepage Merging(KSM), respectively. While the memory saving optimization techniques improve the performance they also create a covert channel that a malicious VM can exploit. In fact, memory disclosure attacks [25] and side-channel attacks [29, 6, 15] have been proposed taking advantage of memory deduplication techniques in the cloud.

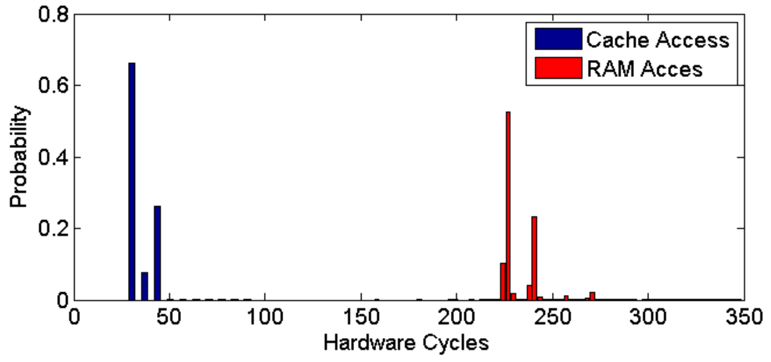


Fig. 1. Data access time in hardware cycles when the data is located in the cache and in the memory

2.3 The *Flush+Reload* Side-Channel attack

The *Flush+Reload* is a trace driven cache side-channel attack that was first used in [12], but acquired its name in [29]. The attack is based on shared memory leakage coming from deduplication processes. One of the main advantages of the *Flush+Reload* spy process is that it does not require the attacker to be core co-resident with the victim and works in a cross-core scenario *as long as a shared last level cache exists*. The attack is carried out in 3 main stages:

Flush stage: In this stage the attacker flushes one or more of the desired memory locations from the cache using the `clflush` command. More precisely, `clflush` evicts the desired memory locations from the entire cache hierarchy, i.e. even from the non-shared cache hierarchies if the last shared level cache is inclusive. Indeed this is the main reason why the attack is applicable across cores.

Victim access stage: In this stage, the attacker waits for sufficient time for the victim to use (or not use) the memory locations that he has flushed in the previous stage.

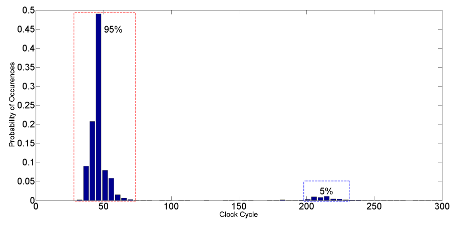
Reloading stage: In the final stage, the attacker reloads the previously flushed memory locations, measuring the reload time for each one of them. If the victim accessed one of the flushed memory lines, due to the inclusiveness of the shared level cache, they will not only be loaded in the upper level caches but also in the shared level cache. Thus, the attacker will measure a lower reload time compared to data accesses to the main memory since the line will be retrieved from the cache. However if the victim did not access to the data flushed in the first stage, the data will still reside in the memory, causing a higher reload time in this reload stage. The different distributions for a memory block accessed from the L1 cache and a memory block accessed from the main memory can be observed in Figure 1. It can be concluded that *Flush+Reload* offers a high distinguishable covert channel due to the significantly different distributions. However, the execution of microarchitectural side-channel techniques can suffer from multiple sources of noise that can be observed in two different ways. The first is a measurement inaccuracy: noise can be introduced by the microarchitecture, by the OS and by the VMM. Often, this noise results in a moderate increase in the number of cycles. However, if e.g. a context switch happens during start and end of a measurement, the value might be off several orders of magnitude. This can be handled by introducing a threshold. Note that such outliers have a significant impact on higher order statistical moments if not filtered out. This said, even with a reasonable threshold, the noise is definitely not Gaussian, possible better described by ExGaussian distributions. The second effect of noise is independent of the measurement process. This happens if a cache line is loaded or evicted by another process. In this case, the source of the timing changes, in addition to the noise introduced during measurement.

3 Attack Description

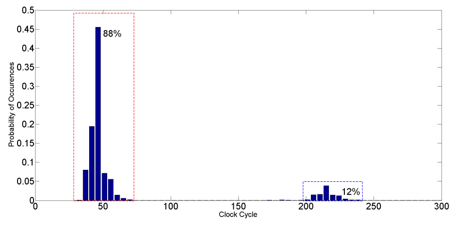
Our attack uses the side-channel technique known as *Flush+Reload* to monitor accesses to memory blocks. The *Flush+Reload* is applicable in the cross-VM setting if deduplication is enabled by the hypervisor and the monitored part of the memory is deduplicated. The latter is true if the monitored data is marked as shared (as is the common case for all crypto libraries) and the hypervisor has detected the duplicated data referenced from within both VMs. Also, different than the attack in [15], we utilize a separate AES detection step to detect the AES execution on the co-located target VM and eliminate the synchronization requirement with the server through the plaintext generation. This makes the proposed attack much more practical. We access the AES function memory address to detect the beginning of AES execution by *Flush+Reload* method. The reason why we access the memory location instead of simply running AES is that accessing a single memory location is much faster than running AES, allowing a higher attack resolution.

3.1 A single cache line attack on AES

The adversary monitors accesses to a single block of one of the T tables used in the last round of AES. In addition to the information t whether the T Table was accessed, the adversary needs to know the corresponding ciphertext c (or plaintext for a first-round attack). That is, we assume the adversary is able to collect several tuples $\langle c, t\rangle$. The monitored memory block corresponds to n T table entries $\mathbb{T}$ known to the adversary. For a monitored ciphertext byte C_i, these entries correspond to n T table outputs S_i, which are mapped one-to-one to n ciphertext byte values through addition with the key. Hence, $c_{i,j} = k_i \oplus s_{i,j}$,



(a) Distribution f_0 for case H_0



(b) Distribution f_1 for case H_1

Fig. 2. 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 in SSA scenario.

where i is a byte position (ignoring the shift rows operation) and j indicates different values. If $s_{i,j}$ is equal to one of the values of the monitored T table memory block, i.e. $s_{i,j} \in \mathbb{T}$, then the monitored memory block will be accessed hence loaded to the cache. We will refer to this case as H_0. However, if $s_{i,j} \notin \mathbb{T}$, i.e. $s_{i,j}$ takes a value stored in a different memory block, then the monitored memory block is not loaded. Nevertheless, since each T table is accessed l times, there is still a high probability that the memory block was loaded by any of the other accesses. In fact, the probability that a memory block is not accessed during an encryption is given as: $\Pr[\text{no access to } T_j] = (1 - {}^{n}\!/_{256})^l$. We will refer to this event as H_1.

For AES-128 in `OpenSSL 1.0.1g`, $n = 16$ and $l = 40$ per T_j, and 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 ϵ_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. To distinguish H_0 from H_1 the *Flush+Reload* method can be applied. In fact, using the *Flush+Reload* method, one can, with high probability, distinguish a cache access from a memory access as seen in Figure 1. In our scenario (as described in Section 4) the leakage distributions f_0 and f_1 are depicted in Figure 2. The distributions are derived from the reload times measured by the *Flush+Reload* attack. The first peak in both distributions (at around 35 cycles) corresponds to a noisy cache reload, and the second peak (at around 220 cycles) corresponds to a memory reload. Since f_0 corresponds to H_0 and hence has more cache reloads than f_1, these distributions are distinguishable. This leakage was successfully exploited in [15].

3.2 Distinguishers for the AES Attack

To process the side-channel data, we describe and compare three distinguishers. The distinguishers we present here analyze one byte of the ciphertext c together with the access time t to the corresponding T table block to recover one byte k of the last round key.

As described earlier, our observations are split into two sets according to a hypothesis. If this hypothesis is correct, the resulting leakage distributions f_0

and f_1 for the two sets differ and hence—with sufficiently many observations—become distinguishable. For wrong key guesses, however, the hypotheses will be invalid, and both sets will sample from the same mixed distribution, making them indistinguishable. To detect whether samples for hypotheses H_0 and H_1 are actually from different distributions, we can apply several distinguishers. In the following we propose three distinguishers. The probably most common distinguisher is based on the difference of the means of the two distributions [20, 11]. As for the zero-value DPA [21], our hypothesis deviates from a single-bit prediction, yet, the test still just distinguishes two cases. Similarly, the variance test uses a statistical moment to distinguish the two distributions [19, 20, 11]. The last distinguisher applies a *miss counter*, as in [15]. The list is neither exhaustive, nor do we make an optimality claim. The latter is interesting future work that needs to be preceded by a better understanding and analysis of the underlying noise characterization, as noise can come from several different sources and is far from being Gaussian.

For the following descriptions we refer to the average miss counter value for H_i as $\overline{ctr}_{H_i}$, whereas we refer to the difference of means and difference of variances for f_i as $\overline{\tau}_{H_i}$ and $\operatorname{var}\tau_{H_i}$, respectively.

Miss-counter based Distinguisher This distinguisher counts and compares the memory block 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 either a hit (0) or a miss (1), depending on whether the value is above or below a threshold access time. As seen in Figure 1, a good threshold for our processor and probing code is 130 cycles. Since H_1 contains significantly more values than H_0, we compare the relative counters instead of absolute ones. 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 the 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 biggest 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. Thus, we use the difference of means distinguisher due to its simplicity.

Variance based Distinguisher The difference of variances distinguisher outputs the difference of variances in cycles.

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

Note that, as before, 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 that can be orders of magnitude larger than regular data are not uncommon and need to be filtered to make this distinguisher work. Since H_i is key dependent, the guessed key $\hat{k}$ that maximizes the difference is the most likely to be correct. 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 a lower miss counter, a lower average (mean) access time, and also a lower variance. The results will show that taking the sign into account derives a much better distinguisher.

When the three distinguishers are compared, the miss counter approach has the most interesting properties: 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.

3.3 Attack Scenarios

Next, we describe the principles of our new *Flush+Reload* attack as well as the original and the improved versions of the attack in [15]. We will refer to the attack in [15] as the Fully Synchronous Attack (FSA) and the improved version of it with the additional AES detection step as the Semi-Synchronous Attack (SSA). Finally, the attack scenario where the attacker requires no synchronization with the server will be referred as the Asynchronous Attack (ASA). In the following, we explain and compare these attacks in detail, listing challenges and advantages of each version.

FSA This is the original attack used in [15] where the attacker first flushes the T tables, then sends a plaintext to the encryption server to trigger an AES encryption. The server receives the plaintext from the attacker, and sends the ciphertext back. Upon receipt of the ciphertext, the attacker reloads the monitored T table blocks to learn which entries were accessed by the encryption.

SSA In this version of the attack, we improved over the FSA by detecting the AES encryption using the *Flush+Reload* method but there is still a need for trigger event by the adversary. The advantage of this attack is the usage of an AES encryption detector that detects whether the victim is performing an AES encryption. Once the AES encryption function call is detected, the attacker flushes the monitored T table blocks **during** the AES execution in between AES rounds. Flushing in between rounds reduces the number $l-1$ of unrelated accesses to the T table accesses, hence increasing the number of memory accesses for case H_1. In addition, we know that the detection algorithm takes half of the timing of an AES encryption. Therefore, at least half of the rounds of the AES encryption is eliminated by this detection mechanism. This results in a more biased distribution f_1, i.e. a stronger leakage. Consequently, the attack succeeds with fewer encryptions.

ASA In the ASA, we improve over the previous two attacks by not requiring any trigger event by the adversary. Instead, plaintexts are generated by the server in regular intervals of 5M cycles. The adversary uses an AES detector to detect the AES function call and perform the *Flush+Reload* attack on the fly. In addition the network is monitored to recover transmitted ciphertexts. Unlike the previous attacks, this attack is a true ciphertext-only attack.

Note that the ASA presents a more realistic attack scenario than those presented in [12] and [15]. In [12] Gullasch et al. described a *Flush+Reload* attack on AES implementation of the `OpenSSL` library where they overload the CPU and suspend the AES encryption by controlling CFS. In [15] authors require synchronization with the server through the plaintext generation. In contrast to these previous attacks, our attack differs in the following ways:

– Our attack flushes the T tables **during** the AES encryption rather than before;
– CFS exploitation or any other type of CPU overloading is not necessary;
– Synchronization through the plaintext is no longer required, but the AES encryption call is detected instead;
– Improved side-channel data analysis/key recovery methods recover the key with fewer encryptions.

4 Experiment Setup

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

- **Native Execution:** In this setup, the AES encryption process and the attacker run on a native Ubuntu 12.04 LTS version with no virtualization. In this setting, we have used a two core Intel i5-2430M CPU clocked at 2.4 GHz. The purpose of this scenario is to run the attack in an environment with minimal noise and to achieve comparability to former non cross-VM cache attacks.
- **Cross-VM Execution:** In this setup, two up-to-date Ubuntu VMs, VM1 and VM2 are launched and managed by VMware ESXI 5.5 baremetal hypervisor. The attacks are then performed across hypervisor isolation boundaries.

The first VM is used as the target that does the AES encryption while the second VM acts as the attacker and executes the *Flush+Reload* attack, trying to recover the secret key. The experiments in this setting were performed on an Intel Xeon E5-2670 v2 CPU. This setup reflects a realistic attack scenario by using a modern CPU commonly used in commercial cloud systems [1, 2]. In this setup, data access from the cache takes 30 cycles and the memory takes 233 cycles on average. Also in the same specification, single AES encryption without and with pre-flushed T-tables requires 257 and 659 cycles, respectively. As the Figure 1 shows, the timing separation between the CPU cache and the main memory is clear with very few outliers. We further observe in Figure 1 that the AES execution time changes greatly depending on whether or not the T-tables used for the encryption are loaded in the cache.

Note that all the timing measurements in the experiments are gathered using the `Read Time Stamp Counter and Processor ID (RDTSCP)` instruction. The usage of the `RDTSCP` instruction is allowed in VMware user mode, but not in KVM. Moreover, this instruction is not emulated by the VMM but executed directly, unlike other serializing instructions like `CPUID` used in [15]. Also, the flushing operation is performed using the `Cache Line Flush (CLFLUSH)` instruction.

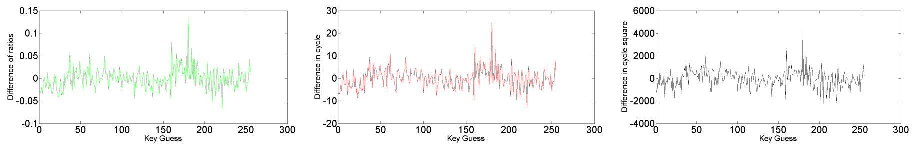
In all experiments, one target process executes AES encryption while the attacker process tries to recover the secret key by monitoring the T-tables with the *Flush+Reload* technique. In order to clearly show the the attack success under different assumptions, we have used two distinct attack environments.

5 Results

We performed the experiments for all three attack scenarios, i.e. FSA, SSA and ASA in both native and virtualized environments. Furthermore, we analyze the timing behavior to show the improvement on the success rate by using the three different distinguishers mentioned in Section 3.2: the miss counter distinguisher, the difference of means distinguisher and the difference of variances distinguisher.

At first we present and compare the scores of the key guesses using the three different distinguishers in native execution in Figure 3. The difference of means and variances distinguishers suffer more from noise due to heavy outliers stemming from different microarchitectural sources of noise. However the experiments shown in Figure 3 were taken cutting off outliers with an outlier threshold value of 5 times the memory access time. It can be seen that for 10,000 encryptions the three distinguishers clearly maximize the score for the correct key, i.e. 180 in this case.

Then, the results of the three different attack scenarios is presented in Table 1 by comparing the ratio between cache accesses and memory accesses for cases H_0 and H_1. The precise distribution for the SSA scenario was given in Figure 2. Recall that without noise, the ratio should be 100%/0% for H_0 vs. 92%/8% for H_1 for the FSA scenario and even more biased for the SSA scenario.



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

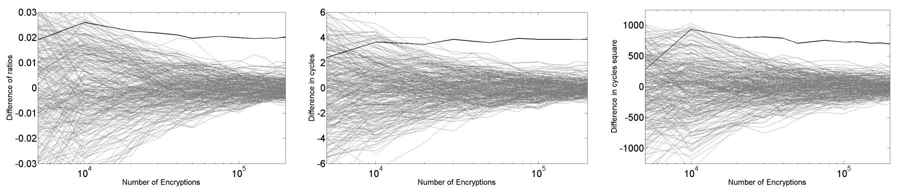
Fig. 3. Comparison of the scores of key guesses in the natively executed FSA scenario for three different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c), applied to 10000 traces. The correct key is 180 and clearly distinguishable in all three cases.

The probability distribution shows that for H_0 approximately 95% of the reload values are coming from L3 cache while only the 5% come from the main memory. In H_1 however, the reload values coming from L3 cache are down to 88%, while the values coming from the main memory increase to 12%. Also, it can be seen from the Table 1 that there is a significant improvement in the distinguishability for SSA scenario due to flushing during AES encryption. Flushing during the encryption translates into lower noise in the T table measured access times and an improved success rate. However, the increased number of detected memory accesses for SSA is likely caused by flushes occurring *after* AES encryption has terminated. Thus, although the more realistic ASA scenario decreases the success rate, in comparison to the SSA scenario due to the difficulty of the AES detection. Hence, SSA is the most efficient way to decrease the noise and have a good resolution to find the correct key with a small number of encryptions.

Table 1. Distribution of cache accesses vs. memory accesses for the two hypotheses over the three attack scenarios. SSA provides the best distinguishability.

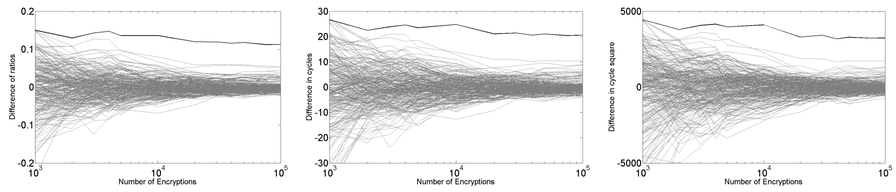
Attack Scenarios	**H_0**		**H_1**	
	Cache	Memory	Cache	Memory
Ideal case	100%	0%	92%	8%
FSA	99%	1%	97%	3%
SSA	95%	5%	88%	12%
ASA	97%	3%	96%	4%

Finally the number of traces needed for the recovery of the key are presented in Figures 4,5. As for the attack scenario success rates, our experiments in the native execution setting show that the SSA yields higher success rate than the FSA and the ASA which require 3,000, 25,000 and 30,000 encryptions ,respectively. Also, the variance distinguisher works better in native setting than the other two distinguishers. For other attack scenarios e.g. the ASA, the mean distinguisher works the best, see Figure 5(b). Note that, since ASA is the most



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

Fig. 4. Comparison of results in native execution for FSA scenario for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c).



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

Fig. 5. Comparison of results in native execution for the SSA for different distinguishers based on the miss counter (a), difference of means (b) and difference of variances (c).

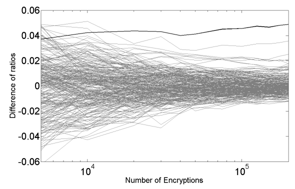
realistic scenario, it requires more encryption samples than the other two, most notably compared to the SSA where only 3,000 encryption samples are needed.

5.1 Cross-VM Execution Results

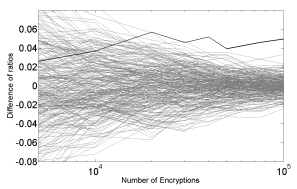
In the cross-VM setting, the FSA scenario requires 30,000 encryptions to recover the full key using the miss counter hypothesis as seen in Figure 6(a). In the same setting, 50,000 encryptions are needed when the difference of means distinguisher is used as in Figure 6(d). As for the SSA, only 10,000 encryptions are enough to recover the full key using the mean distinguisher in Figure 6(b). If the miss counter distinguisher is used instead of the mean distinguisher, 40,000 encryptions are needed as seen in Figure 6(e).

For the ASA scenario, 30,000 encryptions are enough to recover the full key using miss-counter and mean distinguishers as seen in Figures 6(c), 6(f). Also, when we compare different distinguisher methods in the cross-VM setting for different attack scenarios, we see that the difference of means distinguisher works better than the miss-counter distinguisher for the most successful attack which is the SSA. While the miss-counter distinguisher gives better results for the FSA, the two distinguishers have the same impact on the results for the ASA which is the most realistic attack scenario.

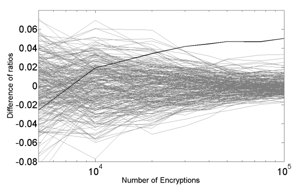
We would like to note that the difference of means and the difference of variances distinguishers work better in the SSA and ASA scenarios, whereas the miss



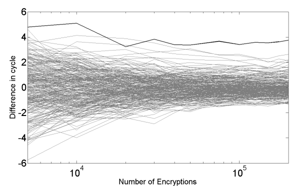
(a) FSA



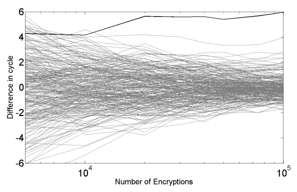
(b) SSA



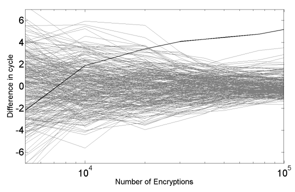
(c) ASA



(d) FSA



(e) SSA



(f) ASA

Fig. 6. Results in cross-VM execution for different attack scenarios using the miss counter distinguisher FSA (a) SSA (b) ASA (c) and the means distinguisher FSA (d) SSA (e) and ASA (f).

counter yields better results for the FSA. Moreover, the main advantage of using the variance and mean distinguishers is that they do not need an architecture dependent threshold, whereas the miss counter approach needs the access time distribution of the cache hierarchy.

Also note that the improvement of SSA is due to flushing **during** the AES execution which yields lower noise in the reloading stage. As for the ASA, we would like to emphasize that the higher number of encryptions requirement is due to the more realistic nature of the attack setting i.e. the lack of synchronization between the server and the spy process. Finally, we would like to remark that **only 15 seconds** are enough to recover the whole key in SSA scenario, which to the best of our knowledge is the fastest working attack in a realistic cross-VM setting without the scheduler exploitation.

6 Conclusion

In conclusion, we would like to remark that in this work, for the first time we have accomplished a cache side-channel attack on AES by flushing in between rounds. We also used an additional AES detection stage to create an asynchronous attack setting. In addition to that, we improved upon the previous work on cross-VM AES attacks by utilizing three different distinguishers for the key recovery. Finally, our experiments show that among three attack scenarios, SSA works with the minimum number of encryptions, requiring only 3,000 in the native and 10,000 in the cross-VM setting.

7 Acknowledgements

This work is supported by the National Science Foundation, under grant CNS-1318919.

References

1. Amazon EC2 Instances. `http://aws.amazon.com/ec2/instance-types/`.
2. Google Compute Engine Instance Types. `https://cloud.google.com/compute/docs/machine-types`.
3. CFS scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, April 2014.
4. Kernel Samepage Merging. `http://kernelnewbies.org/Linux_2_6_32\#head-d3f32e41df508090810388a57efce73f52660ccb/`, April 2014.
5. Aciiçmez, O., and Koç, Ç. K. Trace-driven cache attacks on AES (short paper). In *Information and Communications Security*. Springer, 2006, pp. 112–121.
6. Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *CHES* (2014), pp. 75–92.
7. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.
8. Bonneau, J. Robust Final-Round Cache-Trace Attacks Against AES. *IACR Cryptology ePrint Archive 2006* (2006), 374.
9. Chen Cai-Sen, Wang Tao, C. X.-C., and Ping, Z. An Improved Trace Driven Instruction Cache Timing Attack on RSA. Cryptology ePrint Archive, Report 2011/557, 2011. `http://eprint.iacr.org/`.
10. Craig Labovitz. How Big is Amazons Cloud? `http://www.deepfield.com/2012/04/how-big-is-amazons-cloud/`, 2012.
11. Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., and Willems, J.-L. A Practical Implementation of the Timing Attack. In *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds., vol. 1820 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 167–182.
12. Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. *IEEE Symposium on Security and Privacy 0* (2011), 490–505.
13. Hu, W.-M. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), SP '92, IEEE Computer Society, pp. 52–.
14. Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Fine grain cross-vm attacks on xen and vmware are possible! Cryptology ePrint Archive, Report 2014/248, 2014. `http://eprint.iacr.org/`.
15. Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.
16. Jones, M. T. Inside the linux 2.6 completely fair scheduler. `http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf`, December 2009.
17. Jones, M. T. Anatomy of linux kernel shared memory. `http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/`, April 2010.

18. Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Side Channel Cryptanalysis of Product Ciphers. *J. Comput. Secur. 8*, 2,3 (Aug. 2000), 141–158.
19. Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96* (1996), N. I. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 104–113.
20. Kocher, P. C., Jaffe, J., and Jun, B. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, 1999), Springer-Verlag, pp. 388–397.
21. Mangard, S., Oswald, E., and Popp, T. *Power analysis attacks: Revealing the secrets of smart cards*, vol. 31. Springer, 2008.
22. Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'06, Springer-Verlag, pp. 1–20.
23. Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.
24. Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.
25. Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security* (2011), ACM, p. 1.
26. Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS* (2003), Springer-Verlag, pp. 62–76.
27. VMWare. Understanding Memory Resource Management in VMware vSphere 5.0. `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.
28. Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.
29. Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 719–732.
30. Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 313–328.
31. Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.