Jackpot
Stealing Information From Large Caches via Huge Pages

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar

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

Abstract. The cloud computing infrastructure relies on virtualized servers that provide isolation across guest OS's through sandboxing. This isolation was demonstrated to be imperfect in past work which exploited hardware level information leakages to gain access to sensitive information across co-located virtual machines (VMs). In response virtualization companies and cloud services providers have disabled features such as deduplication to prevent such attacks.
In this work, we introduce a fine-grain cross-core cache attack that exploits access time variations on the last level cache. The attack exploits huge pages to work across VM boundaries without requiring deduplication. No configuration changes on the victim OS are needed, making the attack quite viable. Furthermore, only machine co-location is required, while the target and victim OS can still reside on different cores of the machine. Our new attack is a variation of the prime and probe cache attack whose applicability at the time is limited to L1 cache. In contrast, our attack works in the spirit of the flush and reload attack targeting the shared L3 cache instead. Indeed, by adjusting the huge page size our attack can be customized to work virtually at any cache level/size. We demonstrate the viability of the attack by targeting an `OpenSSL1.0.1f` implementation of AES. The attack recovers AES keys in the cross-VM setting on Xen 4.1 with deduplication disabled, being only slightly less efficient than the flush and reload attack. Given that huge pages are a standard feature enabled in the memory management unit of OS's and that besides co-location no additional assumptions are needed, the attack we present poses a significant risk to existing cloud servers.

Keywords: Cross-VM, huge pages, memory deduplication, prime and probe, flush+reload, cache attacks.

1 Introduction

The end of exponential growth of single core performance in the past decade has helped creating a new industry selling computing infrastructure as a service (IaaS) popularly referred to as *cloud computing*. Instead of financing and maintaining expensive workstations and servers, companies can rent the resources from cloud providers just when the needed and only for the duration of the need thereby significantly cutting IT costs. A number of well-known tech companies such as Google, Amazon AWS, EMC come to mind when mentioning cloud computing. Popular user-oriented examples include cloud backed storage service providers like Dropbox in the personal computing space and Box.net in the enterprise. These are just a couple of examples among numerous businesses enabled by cloud backed compute and storage offerings such as Amazon's EC2 compute and S3 storage solutions, respectively. Nevertheless, like any emerging technology, cloud services have also encountered their unique security challenges. The problem stems from the fact that most security technologies were developed for a world of isolated servers and were subsequently transferred to virtualized servers potentially hosting a number of guest OS's without any adjustments.

A new class of security vulnerabilities arise due to one of the most important principles that cloud systems are based on: co-residency and multi-tenancy. The benefit of could computing comes from resource sharing, implying that multiple customers will utilize the same hardware of the same physical machine instead of assigning a dedicated server per user. Despite the benefits that co-residency bestows, namely maintenance and electricity cost reduction, it also implies that users run their virtual machines (VM) in the same hardware only separated by the virtualization layer provided by the Virtual Machine Manager (VMM). In theory

sandboxing techniques should provide the requested isolation between VMs, but of course *the devil is in the details*.

A serious threat to VM isolation (and therefore the customer's privacy) comes from side channel attacks which exploit subtle information leakage channels at the microarchitectural level. If side channel attacks can circumvent the logical isolation provided by the hypervisor, critical pieces of information such as cryptographic keys might be stolen. In particular, co-residency creates a scenario where microarchitectural side channels can potentially be exploited. A large number of microarchitectural attacks targeting cryptographic keys have already been extensively studied and successfully applied in non-virtualized scenarios. For instance, cache attacks are based on access time variations when retrieving data from the cache and from the memory, as proposed by Bernstein [22] or Osvik et al. [34]. Both studies manage to recover AES secret keys by monitoring the cache utilization. Modern memory saving features like Kernel Samepage Merging (KSM) [18, 13] have also shown to threaten the security of cryptographic processes as proven by Gullasch et.al [26], recovering AES keys with as little as 100 encryptions. However, despite the successful results obtained by the aforementioned attacks in non-virtualized scenarios, still very little research has been done aiming at safe implementation of cryptosystems in the virtualized setting.

It was not until 5 years ago, when motivated by the work done by Ristenpart et al. [36], that the first successful implementations of side channel attacks inside VMs started to appear in the community. In fact, Ristenpart et al. were not only able to co-locate two virtual machines hosted by Amazon EC2 on the same physical hardware, but also managed to recover key strokes used by a victim VM. In consequence, they showed for the first time that side channel attacks can be implemented in the cloud to break through the isolation provided by *sandboxing* techniques.

From that point on, researches have been focusing on recovering fine grain information with new and known side channel techniques targeting weak cryptographic implementations inside VMs, e.g. El Gamal [45] or AES [42]. The *Flush+Reload* technique has been shown to be particularly effective when memory deduplication features are enabled by the VMM. Indeed, Yarom et al. [43] demonstrated attack that recovered RSA keys across VMs running in different cores and hosted by KVM and VMware. Later Irazoqui et al. [30] used the same technique to recover AES keys across VMware VMs. The relevance of these studies is highlighted by the prompt security update by VMware, making memory deduplication an opt-in feature that was formerly enabled by default.

Recognizing the potential for a security compromise, Amazon disabled deduplication on their EC2 compute cloud servers before the attack was demonstrated.

Even though mechanisms that hinder previous attacks have been implemented, the discussion still remains open in the community. Indeed, new side channel attacks (such as the one proposed in this work) compromising the VM isolation techniques may arise, consequently requiring new countermeasures to mitigate the new introduced leakage.

Our Contribution

In this work, we show a novel cross-core and cross-VM cache-based side-channel attack that exploits the shared L3 cache. The attack takes advantage of the additional physical address knowledge gained by the usage of huge size pages. Thus, the attack is not only applicable in non-virtualized environments but also in the cloud, since the huge size pages usage is enabled by default in all common hypervisors, i.e, Xen, VMware and KVM. Unlike the popular *Flush+Reload* attack [43], the new attack does *not* rely on deduplication features (no longer enabled by default in VMware and completely disabled on Amazon AWS servers) and therefore, it can be applied with hypervisors not considered in [43, 30] like Xen. Furthermore, the attack is hardly detectable by the victim, since only a small number of sets are profiled in the L3 cache.

The viability of the new side channel technique is demonstrated by attacking AES in both non-virtualized and virtualized cross VM scenarios. The attack is compared to previous attacks performed on AES in the cloud [29, 30]. The new attack shows to be significantly more efficient than [29] and achieves similar efficiency as [30]. The attack requires a small amount of time to succeed, i.e, the AES key is recovered in less than 3 minutes in fully virtualized Xen 4.1.
In summary, this work

– introduces a new side channel technique targeting the L3 cache enabled by the usage of huge size memory pages.
– Shows that the attack can be applied in the cloud since most of the hypervisors allow the usage of huge size pages by the guest OSs.
– Presents the viability of the new side channel technique by recovering AES keys when attacker and victim are located in different cores.
– Demonstrates that the attack is also practical by recovering the AES key in less than 3 minutes in virtualized settings.

We summarize existing cache-based side-channel attacks as well as virtual address translation and cache addressing in Section 2. The new side channel attack is introduced in Section 3. Results are presented in Section 5. Before concluding in Section 7 possible countermeasures are discussed in Section 6.

2 Background

In this section we give a brief overview of the background needed to understand the new attack presented in this work. After detailing on cache side channel attacks, their history and the improvements that have been developed over the last 15 years a short explanation of Virtual Address Cache Mapping and the previous *Prime+Probe* technique are provided.

2.1 Cache Side Channel Attacks

Cache side channel attacks take advantage of the information leakage coming from microarchitectural time differences when data is retrieved from the cache rather than the memory. The cache is a small memory placed between the CPU and the RAM to avoid the big latency added by the retrieval of the data. Modern processors usually have more than one level of cache to improve the efficiency of memory accesses. Caches base their functionality on two different principles, i.e, *temporal* and *spatial* locality. The first one predicts that data accessed recently will be accessed soon, whereas the latter one predicts that data in nearby locations to the accessed data will also be accessed soon. Thus, when a value is fetched from memory by the CPU, a copy of that value will be placed in the cache, together with nearby memory values to reduce the latency of future accesses.

Obviously, data in cache can be accessed much faster than data only present in memory. This is also true for multilevel caches, where data accessed from the L1 cache will experience lower latencies than data accessed from subsequent cache levels. These time differences are used to decide whether a specific portion of the memory resides in the cache—implying that the data has been accessed recently. The resulting information leakage is particularly harmful for cryptographic algorithms, which might compromise the secret keys involved in their encryption processes. Although many spy processes have been studied targeting the L1 cache, implying core co-location, lately cross-core spy processes have gained most of the attention. In the latter case, typically the last level of cache acts as the covert channel, since it is usually shared by all the cores in most modern processors. Cross-core cache side channel attacks are particularly dangerous in cloud settings, where more than one user co-reside in the same hardware, and the chance of two users being co-located in different cores is significantly high.

Previous Cache Attacks The cache was first considered to be a suitable cover channel for the unauthorized extraction of information in 1992 by Hu [28]. Kesley et al. [31] also mentioned the possibility of cache attacks based on the cache hit/miss ratio. Later, cache attack examples were studied theoretically by Page [35] whereas Tsunoo et al. [39] investigated timing leakage due to internal table look up collisions.

However it was not until 2004 when the first practical implementations of cache attacks were studied. For instance, Bernstein [22] implemented a cache timing attack targeting AES based on the existing microarchitectural leakage when different memory position are loaded in the cache. He used this leakage to recover the full AES key in some implementations. At the same time Osvik et al. [34] investigated the impact of two different trace driven attacks on AES: *Evict + Time* and *Prime+Probe*. They showed that both methods

can be applied in spy processes to recover AES keys. One year later Bonneau and Mironov exploited the cache collisions due to internal table look ups in AES to obtain the secret key [23].

A similar collision timing attack was presented by Acıiçmez et al. [15] targeting the first and second encryption rounds of AES while Neve and Seifert [33] studied the impact of access driven cache attacks in the last round of AES. In 2007 Acıiçmez proved that AES and the data cache were not the only possible target of cache side channel attacks [14]. He discovered leakages in the instruction cache during public key encryptions and applied cache side channel attacks to recover RSA keys.

However, most of the above mentioned attacks were implemented as spy processes in a native OS environment, reducing the practical impact of the attacks in realistic scenarios. It was not until 2009 when Ristenpart et al. proved to be able to co-locate two virtual machines in a public cloud, achieving the usage of the same CPU [36]. Their experiments in the Amazon EC2 public cloud [2] show that they achieved up to 40% co-residency success rate with the desired target by using different properties like IP range and instance type. The work also demonstrated that cache usage can be analyzed to deduce secret keystrokes used by a potential victim. Hence, the attack demonstrated for the first time that microarchitectural side channel attacks that require co-location are a potential threat in the cloud setting. Further co-residency detection methods such as traffic analysis later were studied, e.g. by Bates et al. [20].

The research made on detecting co-residency motivated many researchers to apply known side channel techniques in the cloud. For instance, Zhang et al. [44] used the above mentioned *Prime+Probe* technique to detect whether any other tenant was co-located in the same hardware. Shortly later again Zhang et al. [45] recovered El Gamal encryption keys by monitoring the L1 instruction cache in a virtualized setting, again with the *Prime+Probe* spy process. Their experiments were carried out in Xen VMs and they had to apply a hidden Markov model to reduce the noise present in their measurements. Bernstein's attack was also tried in virtualized environments, first by Weiss et al. [42] in ARM processors and then by Irazoqui et al. in VMware or Xen [29].

At the same time new spy processes and improvements over previous techniques were investigated in non-virtualized scenarios. Chen et al. presented an improvement over Acıiçmez's technique to monitor the instruction cache and recover a RSA key [24], whereas Aly et al. [17] studied an improvement on the detection method for the Bernstein's attack. Cache collision attacks on AES and instruction cache attacks on DSA were also further investigated by Spreitzer and Plos [37] and Acıiçmez et al. [16] respectively. In the other hand Gullasch et al. [26] studied a new side channel technique that would later acquire the name of *Flush+Reload* and that is based on memory saving features like Kernel Samepage Merging (KSM). They were able to recover a full AES key by monitoring the data cache while getting control of the Control Fair Scheduler (CFS) [12]. This new method proved that successful cache attacks can still be implemented in modern processors, contrary to the statement made in [32].

More recently, Yarom et al. used the *Flush+Reload* technique to attack the RSA implementation of `Libgcrypt` [43]. Furthermore, they showed that their attack is applicable in a cross-core and cross-VM setting. Hence, it could be applied in cloud environments, particularly in the VMMs implementing memory deduplication features like VMware or KVM. Shortly later, Benger et al. applied the same technique to recover ECDSA keys [21]. Finally, Irazoqui et al. demonstrated that *Flush+Reload* can be applied tor recover AES keys without the need of controlling the CFS, and also proved the viability of their method across VMware VMs [30].

2.2 Virtual Address Translation and Cache Addressing

In this work we present an attack that takes advantage of some known information in the virtual to physical address mapping process. Thus, we give a brief overview about the procedure followed by modern processors to access and address data in the cache [27].

In modern computing, processes use *virtual memory* to access the different requested memory locations. Indeed processes do not have direct access to the physical memory, but use virtual addresses that are then mapped to physical addresses by the Memory Management Unit (MMU). This virtual address space is managed bv the Operating System. The main benefits of virtual memory are security (processes are isolated from *real memory*) and the usage of more memory than physically available due to paging techniques.

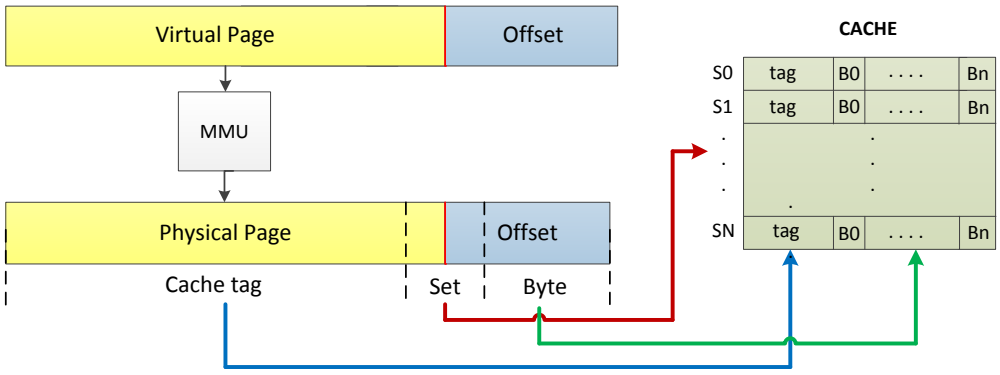

Fig. 1. Cache accesses when it is physically addressed.

In fact, the memory is divided into fixed length continuous blocks called memory pages. The virtual memory allows the usage of these memory pages even when they are not allocated in the main memory. When a specific process needs a page not present in the main memory, a page fault occurs and the page has to be loaded from the auxiliary disk storage. Therefore, a translation stage is needed to map virtual addresses to physical addresses prior to the memory access. In fact, cloud systems have two translation processes, i.e, guest OS to VMM virtual address and VMM virtual address to physical address. The first translation is handled by shadow page tables while the second one is handled by the MMU. This adds an abstraction layer with the physical memory that is handled by the VMM.

During translation, the virtual address is split into two fields: the offset field and the page field. The length of both fields depends directly on the *page size*. Indeed, if the page size is p bytes, the lower $\log_2(p)$ bits of the virtual address will be considered as the *page offset*, while the rest will be considered as the *page number*. Only the page number is processed by the MMU and needs to be translated from virtual to physical page number. The page offset remains untouched and will have the same value for both the physical and virtual address. Thus, the user still knows some bits of the physical address. Modern processors usually work with 4 KB pages and 48 bit virtual addresses, yielding a 12 bit offset and the remaining bits as virtual page number.

In order to avoid the latency of virtual to physical address translation, modern architectures include a Translation Lookaside Buffer (TLB) that holds the most recently translated addresses. The TLB acts like a small cache that is first checked prior to the MMU. One way to avoid TLB misses for large data processes is to increase the *page size* so that the memory is divided in less pages [25, 4, 41]. Since the possible virtual to physical translation tags have been significantly reduced, the CPU will observe less TLB misses than with 4 KB pages. This is the reason why most modern processors include the possibility to use huge size pages, which typically have a size of at least 1 MB. This feature is particularly effective in virtualized settings, where virtual machines are typically rented to avoid the intensive hardware resource consumption in the customers private computers. In fact, most well known VMMs support the usage of huge size pages by guest OSs to improve the performance of those heavy load processes [9, 5, 10].

Cache Addressing: Caches are physically tagged, i.e, the physical address is used to decide the position that the data is going to occupy in the cache. With b bytes size cache lines and m-way set associative caches (with n number of sets), the lower $\log_2(b)$ bits of the physical address are used to index the byte in a cache line, while the following $\log_2(n)$ bits select the set that the memory line is mapped to in the cache. A graphical

example of the procedure carried out to address the data in the cache can be seen in Figure 1. Recall that if a page size of 4 KB is used, the offset field is 12 bits long. If $\log_2(n) + \log_2(b)$ is not bigger than 12, the set that a cache line is going to occupy can be addressed by the offset. In this case we say that the cache is virtually addressed, since the position occupied by a cache line can be determined by the virtual address. In contrast, if more than 12 bits are needed to address the corresponding set, we say that the cache is physically addressed, since only the physical address can determine the location of a cache line. Note that when huge size pages are used, the offset field is longer, and therefore *bigger* caches can be virtually addressed. As we will see, this information can be used to mount a cross-VM attack in the L3 cache.

2.3 The *Prime+Probe* Technique

The new attack proposed in this work is based on the methodology of the known *Prime+Probe* technique. *Prime+Probe* is a cache-based side channel attack technique used in [34, 44, 45] that can be classified as an access driven cache attack. The spy process ascertains which of the sets have been accessed in the cache by a victim. The attack is carried out in 3 stages:

- **Priming stage:** In this stage, the attacker fills the monitored cache with own cache lines. This is done by reading own data.
- **Victim accessing stage:** In this stage the attacker waits for the victim to access some positions in the cache, causing the eviction of some of the cache lines that were primed in the first stage.
- **Probing stage:** In this stage the attacker accesses the priming data again. When the attacker reloads data from a set that has been used by the victim, some of the primed cache lines have been evicted, causing a higher probe time. However if the victim did not use any of the cache lines in a monitored set, all the primed cache lines will still reside in the cache causing a low probe time.

The *Prime+Probe* side channel attack has some limitations. First, it can only be applied in small caches (typically the L1 cache), since only a few bits of the virtual address are known. Second, the application of such a spy process in small caches restricts its application to core co-located processes. Finally, modern processors haver very similar access times for L1 and L2 caches, only differing in a few cycles, which makes the detection method noisy and difficult. This difficulty is indicated e.g. in [45], where the authors had to apply a Hidden Markov Model in addition to the *Prime+Probe* technique to deal with noisy measurements.

3 The *S$A* attack

In this section we present the technical details of our *S$A* attack. Later we demonstrate the viability of the attack on the `OpenSSL1.0.1.f`'s C-implementation of AES [1] to achieve a full AES key recovery in a scenario where attacker and victim are co-located in different cores. Our *S$A* attack has several advantages over the previous cache side channel attacks on AES:

- Our *S$A* attack is the *first* efficient cross-core cache attack that does not take advantage of deduplication processes, yet succeeds in retrieving key information across VM boundaries. While some previous attacks like *Flush+Reload* rely on deduplication features, other attacks like *Prime+Probe* were also applied in the cloud but assumed to be co-located in the same core with the target process. In contrast, the new *S$A* attack detects accesses made to the last level cache by using huge size pages to allocate the attackers data. Since the last level of cache is usually shared among all the cores in modern processors, our spy process can detect cache accesses even when the victim is co-located in a different core on the same machine;
- We almost achieve the same efficiency as the *Flush+Reload* attack with the *S$A* spy process. Other attacks like Bernstein's attack require a much higher number of encryptions to get partial information of the AES key;
- The *S$A* can be considered a non-intrusive cache attack. In the case of AES only 4 sets from the last level cache need to be monitored to recover a full AES encryption key.

3.1 *S$A* enabled by Huge Pages

The *S$A* attack proposed in this work, is enabled by making use of Huge pages and thereby eliminating a major obstacle that normally restricts the *Prime+Probe* attack to target the L1 cache. As explained in Section 2, a user does not use the physical memory directly, but he is assigned a virtual memory so that a translation stage is performed from virtual to physical memory at the hardware level. The address translation step creates a additional challenge to the attacker since real addresses of the variables of the target process are unknown to him. However this translation is only performed in some of the higher order bits of the virtual address, while a lower portion, named the *offset*, remains untouched. Since caches are addressed by the physical address, if we have cache line size of b bytes, the lower $\log_2(b)$ bits of the address will be used to resolve the corresponding byte in the cache line. Furthermore if the cache is set-associative and for instance divided into n sets, then the next $\log_2(n)$ bits of the address will select the set that each memory data is going to occupy in the cache. The $\log_2(b)$-bits that form the byte address within a cache line, are contained within the offset field. However, depending on the cache size the following field which contains the set address may exceed the offset boundary. The offsets allow addressing within a memory page. The OS's Memory Management Unit (MMU) keeps track of which page belongs to which process. The page size can be adjusted to better match the needs of the application. Smaller pages require more time for the MMU to resolve.

Here we focus on the default 4 KB page size and the larger page sizes provided under the common name of Huge pages. As we shall see, the choice of page size will make a significant difference in the attackers ability to carry out a successful attack on a particular cache level:

- **4 KB pages:** For 4 KB pages, the lower 12-bit offset of the virtual address is not translated while the remaining bits are forwarded to the Memory Management Unit. In modern processors the memory line size is usually set as 64 bytes. This leaves 6 bits untouched by the Memory Management Unit while translating regular pages. As shown in the top of Figure 2 the page offset is known to the attacker. Therefore, the attacker knows the 6-bit byte address plus 6 additional bits that can only resolve accesses to small size caches (64 sets at most). This is the main reason why techniques such as *Prime+Probe* have only targeted the L1 cache, since it is the only one permitting the attacker to have full control of the bits resolving the set. Therefore, the small page size indirectly prevents attacks targeting big size caches like the L2 and L3 caches.
- **Huge pages:** The scenario is different if we work with huge size pages. Typical huge page sizes are at 1 MB or even greater. This means that the offset field in the page translation process is bigger, with 20 bits or more remaining untouched during page translation. Observe the example presented in Figure 2. For instance, assume that our computer has 3 levels of cache, with the last one shared, and that 64, 512 and 4096 are the number of sets the L1, L2 and L3 caches are divided into, respectively. The first lowest 6-bits of the offset are used for addressing the 64 byte long cache lines. The following 6 bits are used to resolve the set addresses in the L1 cache. For the L2 and L3 caches this field is 9 and 12-bits wide, respectively. In this case, a huge page size of 256 KB (18 bit offset) or higher will give the attacker full control of the set occupied by his data in all three levels of cache, i.e. L1, L2 and L3 caches. A 256 KB or higher page size, will enable an attacker to target individual lines of the entire L3 cache. The significance of targeting last level cache becomes apparent when one considers the access time gap between the last level cache and the memory, which is much more pronounced compared to the access time difference between the L1 and L2 caches. Therefore, using huge pages makes it possible to reach a higher resolution *Prime+Probe* style attack.

3.2 The *S$A* Attack

The *S$A* technique takes advantage of the control of the lower k bits in the virtual address that we gain with the huge size pages. These are the main steps that our spy process will follow to detect accesses to the last level cache:

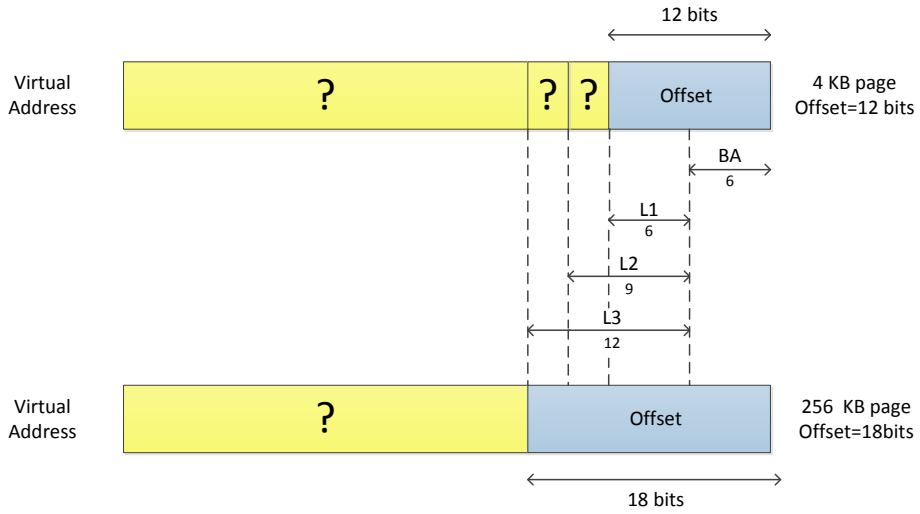

Fig. 2. Regular Page (4 KB, top) and Huge Page (256 KB, bottom) virtual to physical address mapping for an Intel x86 processor. For Huge pages the entire L3 cache sets become transparently accessible even with virtual addressing.

- **Step 1 Allocation of huge size pages:** The spy process is based on the control that the attacker gains on the virtual address when using huge size pages. Therefore the spy process has to have access to the available huge pages, which requires administrator rights. Recall that this is not a problem in the cloud scenario where the attacker is the owner of the administrator right of his OS.
- **Step 2 Prime the desired set in the last level cache:** In this step the attacker creates data that will occupy one of the sets in the last level cache. By controlling the virtual address, the attacker knows the set that the created data is going to occupy in the last level cache. Once enough lines are created to occupy the set, the attacker primes it and ensures that the set is filled. Typically the last level caches are inclusive. Thus we will not only fill the shared last level cache set but also some sets in the upper level caches.
- **Step 3 Reprime to ensure that our data only resides in last level cache:** Priming all cache levels can lead to wrong predictions due to the different access times between the last level of cache and the upper levels. Since we clearly want to distinguish between accesses from the last level cache and memory, we reprime our upper level caches. The basic idea is to be sure to evict our data from the upper level caches, but not from the last level cache. Therefore we ensure that our reprime data goes to a *different* set in the last level cache, but to the same set in the upper level caches.
- **Step 4: Victim process runs:** After the two priming stages, the victim runs the target process. Since one of the sets in the last level cache is already filled, if the targeted process uses the monitored set, one of the primed lines is going to be evicted. Remember we are priming the last level cache, so evictions will cause memory lines to reside in the memory. If the monitored set is not used, all the primed lines are going to reside in the last level cache after the victim's process execution.
- **Step 5: Probe and measure:** Once the victim's process has finished, the spy process probes the primed memory lines and measures the time to probe them all. If one or more lines have been evicted by the targeted process, they will be loaded from the memory and we will see a higher probe time. However if all the lines still reside in the set, then we will see a shorter probe time.

The last step can be made more concrete with the experiment results summarized in Figure 3. The experiment was performed in native execution (no VM) in a intel i5-3320M that has a 16-way associative

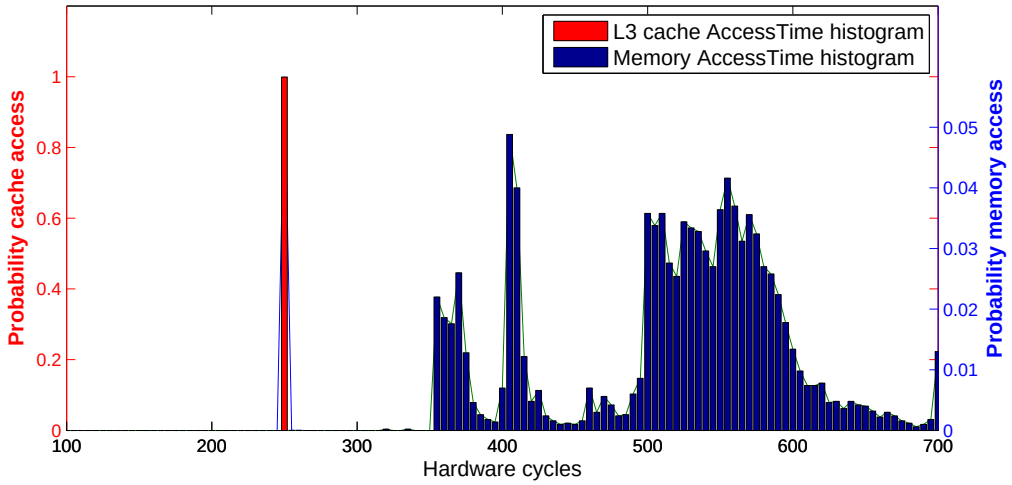

Fig. 3. Histograms of 10,000 access times in the probe stage when all the lines are in the L3 cache and when all except one are in the cache (and the other one in the memory).

last level cache. It can be seen that when all the lines reside in the last level cache we obtain very precise probe timings with average around 250 cycles and with very little variance. However when one of the lines is evicted from last level cache and resides in memory, both the access time and the variance are higher. We conclude that both types of accesses are clearly distinguishable.

For further clarification of the prime and reprime stages we present an example in Figure 4. Assume that we want to monitor set 0 in the last level cache. The last level cache has 1024 sets, and the upper level caches have only 64 sets. Assume that the associativity for this cache is 8 and 4 respectively for the last level cache and the upper level caches, and that the memory line size is 64 bytes. In the example we also assume that all the caches are inclusive. We know that bits $0-5$ will select the corresponding byte in the memory line. We set our data so that the virtual address is 0 from bit 6 to bit 15, in order to ensure that we are filling set 0 in the last level cache. We have to take into account that not only the last level cache will be filled, but also the upper level caches. The reprime stage evicts the blue lines in the upper level caches and replaces them with the yellow lines, which will go to a different set in the last level cache. With this technique we ensure that the lines we are working with only reside in set 0 of the last level cache.

Handling Cache Slices: Typically the last level of cache is divided into *slices* [46, 7, 25]. This means that if the specifications say that we have a 4 MB last level cache, this might be divided into two (or more) slices of 2 MB each. Suppose now that the last level cache is a m-way set associative cache, and that it has n sets. If the last level cache is divided into two slices, we would be addressing $n/2$ sets instead of n sets. Depending on the slice selection method that the architecture implements, our data occupies slice 0 or slice 1. Recall that the last level of cache is usually shared among all the cores. This means that if the cache is not divided into slices, two cores will not be able to access data in the same set in the same clock cycle. However if the cache is divided in two slices, there is a 50% chance that two different cores are accessing different slices and therefore, can access data in the same set in the same clock cycle.

The division of the last level of cache in slices implies an additional step in the execution of the *S$A*. Depending on the algorithm used to select the corresponding slice, the selection of the lines that fill one of the sets of one of the slices can be difficult. However we can always identify the lines that fill a specific set in a slice by measuring the reload time of those lines. If we are working with an m-way associative cache, we need m lines to fill one of the sets in one of the slices. We can verify that we found those specific lines when priming and probing $m+1$ lines gives a significantly higher reload time, since the $(m+1)^{th}$ line evicts one

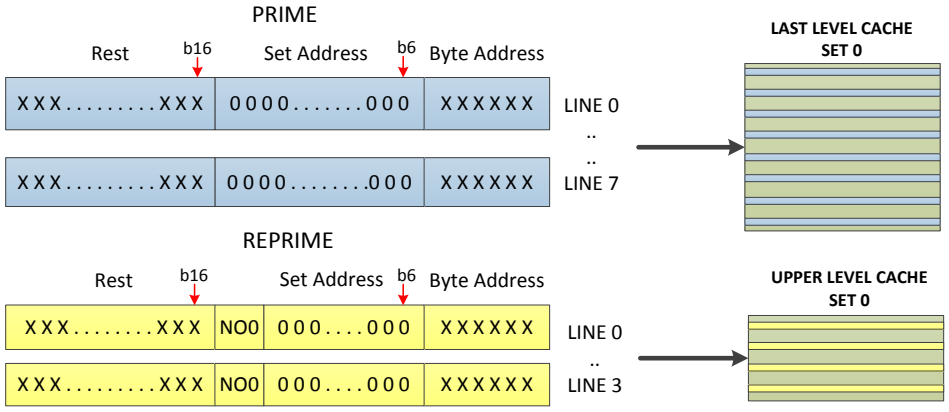

Fig. 4. Prime and reprime stages to ensure we monitor the last level cache.

of the previous ones. Using this method, it is straightforward to try and identify such cache lines for each slice.

The Intel i5-MM30 processor used in our experiments has a two-sliced last level cache. The slice where the data is going to be located is selected with $(l+1)^{th}$ bit, assuming we have l bits to address the set and cache line byte. If the $(l+1)^{th}$ bit is 0, the data will be stored in slice number 0, whereas if the bit is a 1, the data will be stored in the slice number 1.

4 *S$A* applied to AES

In this section we proceed to explain how the *S$A* spy process can be applied to attack AES. We use the C reference implementation of `OpenSSL1.0.1f` library which uses 4 different T-tables during the AES execution. The implementation of AES is based on the execution of three main operations, i.e., a Table lookup operation, a MixColumns operation and a key addition operation. For AES-128 these operations are repeatedly executed for 9 rounds, whereas the last round only implements the Table look up and key addition operations. `OpenSSL` uses 4 different 1 KB size T-tables for the 10 rounds. Recovering one round key is sufficient for AES-128, as the key scheduling is invertible.

We use the last round as our targeted round for convenience. Since the 10^{th} round does not implement the MixColumns operation, the ciphertext directly depends on the T-table position accessed and the last round key. Assume S_i to be the value of the ith byte prior to the last round T-table look up operation. Then the ciphertext byte C_i will be:

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

where T_j is the corresponding T-table applied to the i^{th} byte and K_i^{10}. It can be observed that if the ciphertext and the T-table positions are known, we can guess the key by a simple xor operation. We assume the ciphertext to be always known by the attacker. Therefore the attacker will use the *S$A* spy process to guess the T-table position that has been used in the encryption and consequently, obtain the key.

Since *S$A* will decide which table look up position has been used by monitoring memory accesses, we need to know how the T-tables are handled in memory. With 64 byte memory lines, each T-table occupies 16 cache lines and each cache line holds 16 T-table positions for `OpenSSL 1.0.1f`. 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:

$$\text{Prob[no access}T[i]] = (1 - (15/16))^{40} \approx 8\%. \tag{2}$$

Thus, if the attacker knows which set each of the T-table memory lines occupy, *S$A* will detect that the set is not accessed 8% of the times. We use the same procedure as in [30] to determine the key used in the last round operation. Each ciphertext value is going to be assigned a counter that will depend on the usage of the monitored T-table line. Recall that the usage of the monitored T-table memory line could have happened in any of the 10 rounds of AES. However, since the accesses are profiled according to the corresponding ciphertext value, the attacker has two options:

- **Assign an *access counter*:** Assign an access counter to each possible ciphertext byte value C_i that increments each time the monitored T-table line *is accessed*. In this scenario, once enough measurements have been taken, the ciphertext values corresponding to the monitored T-table line will present higher counters than the rest.
- **Assign a *miss counter*:** Assign a miss counter to each possible ciphertext byte value C_i that increments each time the monitored T-table line *is not accessed*. Thus, once enough measurements have been taken, the ciphertext values corresponding to the monitored T-table line will present minimum values.

Measuring microarchitectural timings implies dealing with noise that increases the measured time, e.g., TLB misses and context switches. Since in our attack scenario this noise is most of the time only biased in one direction (increasing access times), we decide to use the *miss counter*, since it is less susceptible to noise.

Thus, once enough measurements have been done by *S$A* we will see that 16 ciphertext values have significantly higher access counters than the rest. The key is obtained by solving Equation (1), i.e, xoring each of the ciphertext values with each of the values in the monitored T-table memory line. This operation outputs sets of possible keys for each ciphertext value, while the correct key is present in all of them

Locating the Set of the T-Tables: The previous description implicitly assumes the attacker to know the location, i.e. the sets, that each T-table occupies in the shared level cache. A simple approach to gain this knowledge is to prime and probe every set in the cache, and analyze the timing behavior for a few random AES encryptions. The T-table based AES leaves a distinctive fingerprint on the cache, as T-table size as well as the access frequency (92% per line per execution) are known. Once the T-tables are detected, the attack can be performed on a single line per table. Nevertheless, this locating process can take a significant amount of time when the number of sets is sufficiently high in the outermost shared cache.

An alternative, more efficient approach is to take advantage of the shared library page alignment that some OSs like Linux implement. Assuming that the victim is not using huge size pages for the encryption process, the shared library is aligned with a 4 KB page boundary. This gives us some information to narrow down the search space, since the lower 12 bits of the virtual address will not be translated. Thus, we know the offset f_i modulo 64 of each T-table memory line and the T-table location process has been reduced by a factor of 64. Furthermore, we only have to locate one T-table memory line per *memory page*, since the remaining table occupies the consecutive sets in the last level cache.

Attack stages: Putting all together, these are the main stages that the we follow to attack AES with *S$A*

- **Step 1: Last level cache profile stage:** The first stage to perform the attack is to gain knowledge about the structure of the last level cache, the number of slices, and the lines that fill one of the sets in the last level cache.
- **Step 2: T-table set location stage:** The attacker has to know which set in the last level cache each T-table occupies, since these are the sets that need to be primed to obtain the key.
- **Step 3: Measurement stage:** The attacker primes and reprimes the desired sets, requests encryptions and probes again to check whether the monitored sets have been used or not.
- **Step 4: Key recovery stage:** Finally, the attacker utilizes the measurements taken in Step 3 to derive the last round key used by the AES server.

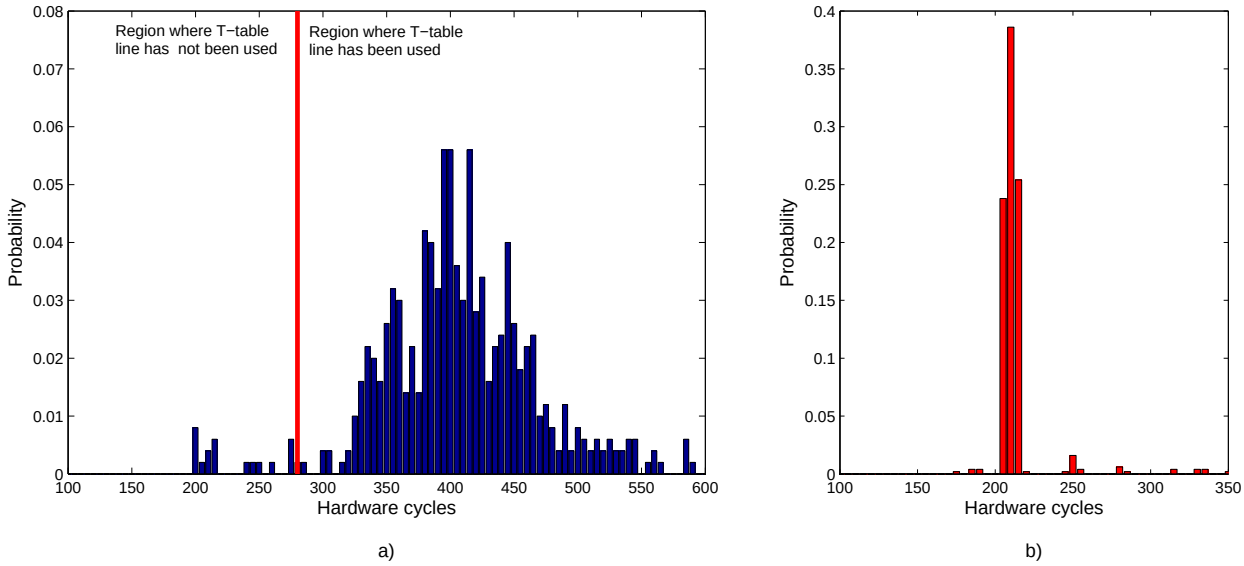

Fig. 5. Histograms of 500 access times monitored in the probe stage for a) a set used by a T-table memory line and b) a set not used by a T-able memory line. Measurements are taken in the Xen 4.1 cross-VM scenario.

5 Experiment Setup and Results

In this section we analyze our experiment setup and the results obtained in native machine, single VM and in the cross-VM scenarios. We also include a comparison with previous attacks that were performed in virtualized scenarios targeting AES.

5.1 Testbed Setup

The machine used for all our experiments is a dual core Intel i5-3320M [6] clocked at 3.2 GHz. This machine works with 64 byte cache lines and has private 8-way associative L1 and L2 caches of size 2^{15} and 2^{18} bytes, respectively. In contrast, the 16-way associative L3 cache is shared among all the cores and has a size of 2^{22} bytes, divided into two slices. Consequently, the L3 cache will have 2^{12} sets in total. Therefore 6 bits are needed to address the byte address in a cache line and 12 more bits to specify the set in the L3 cache. The huge page size is set to 2 MB, which ensures a set field length of 21 bits that are untouched in the virtual to physical address translation stage. All the guest OSs use Ubuntu 12.04, while the VMM used in our cloud experiments is Xen 4.1 fully virtualized, which allows the usage of huge size pages by guest OSs [19, 10, 11].

The target process is going to use the C reference implementation of `OpenSSL1.0.1f`, which is the default if the library is configured with `no-asm` and `no-hw` options. We would like to remark that these are not the default OpenSSL installation options in most of the products.

The attack scenario is going to be the same one as in [22, 30], where one process/VM is handling encryption requests with an secret key. The attacker's process/VM is co-located with the encryption server, but in different cores. We assume synchronization with the server, i.e, the attacker starts the *S$A* spy process and then sends random plaintexts to the encryption server. The communication between encryption server and attacker is carried out via socket connections. Upon the reception of the ciphertext, the attacker measures the L3 cache usage by the *S$A* spy process. All measurements are taken by the attackers process/VM with the `rdtscp` function, which not only reads the time stamp counters but also ensures that all previous processes have finished before its execution [3].

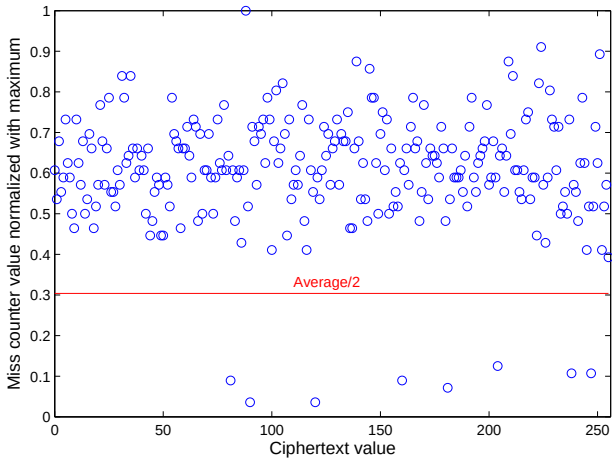

Fig. 6. Miss counter values for ciphertext 0 normalized to the maximum value. The key is e1 and we are monitoring the last 8 values of the T-table (since the table starts in the middle of a memory line).

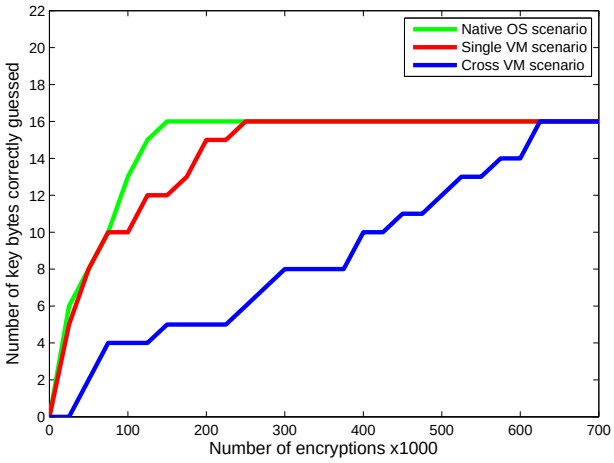

Fig. 7. Number of key bytes correctly recovered vs number of encryptions needed for native OS, single VM and cross-VM scenarios.

5.2 The Cross-Core Cross-VM Attack

We perform the attack in three different scenarios: native machine, single VM and cross-VM. In the native and single VM scenarios, we assume that the huge size pages are set to be used by any non-root process running in the OS. Recall that in the cross-VM scenario, the attacker itself has administrator rights in his own OS.

The first step is to recognize the access pattern of the L3 cache in our Intel i5-3320M. Using *S$A* we detect that the L3 cache is divided in more than one slice, since generating 17 random lines that occupy the set 0 in the cache does not output higher probe timings. The spy process helps us to understand that the cache is divided into two slices, and that the slice selection method is based on the parity of the 17th bit, i.e, the first non set addressing bit. Thus we need 16 odd lines to fill a set in the odd slice, whereas we need 16 even lines to fill a specific set in the even slice.

The second step is to recognize the set that each T-table cache line occupies in the L3 cache. For that purpose we monitor each of the possible sets according to the offset obtained from the linux shared library alignment feature. Recall that if the offset modulo 64 f_0 of one of the T-tables is known, we only need check the sets that are 64 positions apart, starting from f_0. By sending random plaintexts the set holding a T-table cache line is used around 90% of the times, while around 10% of the times the set will remain unused. The difference between a set allocating a T-table cache line and a set not allocating a T-table cache line can be graphically seen in Figure 5, where 500 random encryptions were monitored with *S$A* for both cases in a cross-VM scenario in Xen 4.1. It can be observed that monitoring an unused set results in more stable timings in the range of 200-300 cycles. However monitoring a set used by the T-tables outputs higher time values around 90% of the time, whereas we still see some lower time values below 300 around 10% of the times. Note that the key used by the AES server is irrelevant in this step, since the set used by the T-table cache lines is going to be independent of the key.

The last step is to run *S$A* to recover the AES key used by the AES server. We consider as valid ciphertexts for the key recovery step those that are at least below half the average of the overall timings. This threshold is based on empirical results that can be seen in Figure 6. The figure presents the miss counter value for all the possible ciphertext values of C_0, when the last line in the corresponding T-table is monitored. The key in this case is $0xe1$ and the measurements are taken in a cross-VM scenario in Xen 4.1. In this case

Table 1. Comparison of cross-VM cache side-channel attacks on AES

Attack	Platform	Methodology	`OpenSSL`	Traces
Spy-Process based Attacks:				
Collision timing [23]	Pentium 4E	Time measurement	`0.9.8a`[1]	300.000
Prime+probe [34]	Pentium 4E	L1 cache prime-probing	`0.9.8a`	16.000
Evict+time [34]	Athlon 64	L1 cache evicting	`0.9.8a`	500.000
Flush+Reload (CFS)[2][26]	Pentium M	*Flush+reload* w/CFS	`0.9.8m`	100
Flush+Reload [30]	i5-3320M	L3 cache *Flush+reload*	`0.9.8a`	8.000
Bernstein [17]	Core2Duo	Time measurement	`1.0.1c`	2^{22}
Flush+Reload [30]	i5-3320M	L3 cache *Flush+reload*	`1.0.1f`	100.000
S$A[3]	i5-3320M	L3 cache *S$A*	`1.0.1f`	150.000
Cross-VM Attacks:				
Bernstein [29][4]	i5-3320M	Time measurement	`1.0.1f`	2^{30}
Flush+Reload (VMware)[5][30]	i5-3320M	L3 cache *Flush+Reload*	`1.0.1f`	400.000
S$A (Xen)	i5-3320M	L3 cache *S$A*	`1.0.1f`	650.000

[1] `OpenSSL 0.9.8a` uses a less noisier implementation.
[2] The attack is performed taking control of the CFS.
[3] Huge Pages have to be configured to allow non-root processes to use them.
[4] Only parts of the key were recovered, not the whole key.
[5] The attack is only possible if deduplication is enabled by the VMM. Transparent Page Sharing is no longer enabled by default in VMware. Amazon disabled deduplication on all their AWS servers.

only 8 values take low miss counter values because the T-table finishes in the middle of a cache line. These values are clearly distinguishable from the rest and appear in opposite sides of the empirical threshold.

Results for the three scenarios are presented in Figure 7, where it can be observed that the noisier the scenario is, e.g. in the cross-VM scenario, the more number of encryptions are needed to recover the key. The plot shows the number of correctly guessed key bytes vs. the number of encryptions needed. Recall that the maximum number of correctly guessed key bytes is 16 for AES-128. The attack only needs 150.000 encryptions to succeed on recovering the full AES key in the native OS scenario. Due to the higher noise in the cloud setting, the single VM recovers the full key with 250.000 encryptions whereas cross-VM scenario requires 650.000 encryptions to recover the 16 key bytes. It is important to remark that the attack is completed in only 9, 30 and 150 seconds respectively in each of the scenarios. A significant portion of these delays stems from the fact that we are using a fully virtualized hypervisor in the single and cross-VM scenarios. Moreover, in the cross-VM scenario the external IP communication adds significant latency.

5.3 Comparison with previous attacks

We compare the efficiency of the attack presented in this work with previously proposed attacks that targeted. The comparison is presented in Table 1. We make the following observations:

- Our attack is close to the efficiency achieved by the *Flush+Reload* attack in non-virtualized environments, and improves over previously proposed attacks. However, huge pages are required to be configured so that their usage by non-root processes is allowed.
- Our new *S$A* attack is more efficient than Bernstein's attack in the cloud, which does not recover the entire key in the cloud even with a significantly higher number of encryptions.
- In the cloud, *S$A* again requires more encryptions than *Flush+Reload* but not as much as to become impractical. The attack can still be realized under 3 minutes. However, it should be noted that *S$A* does not take advantage of memory deduplication process which is crucial for the cross-VM *Flush+Reload* attack. The deduplication feature (called Transparent Page Sharing in VMware) is now disabled by

default in VMware [8]. Moreover, Amazon Web Services confirmed that deduplication features have never been implemented in their public cloud system.

Thus, the *S$A* attack turns VMMs that are not vulnerable to *Flush+Reload* due to the lack of memory deduplication features such as Xen have into a valid target for cross-VM attacks. The only requirement is that guest OSs are allowed to use huge size pages. This feature is implemented at the OS level, and is not administered by the VMM.

6 Applicability and Countermeasures

In this section we shortly comment on the applicability of this attack beyond the scope of AES software implementations and discuss ways how this attack can be prevented.

6.1 Applicability of *S$A*

As described earlier, the *S$A* attack is a cross-core cross-VM attack. *S$A* targets the shared level of cache (typically L3) in a SMP multiprocessor, hence can be used across cores. That is, the attack works even if victim and spy are running on different cores in the same CPU. Unlike other cross-VM attacks, *S$A* does not require deduplication of the targeted data. Previous attacks use deduplication to solve two independent problems. The obvious one is the detection of cache accesses to extract secret information of the victim. However, deduplication also solves the location problem, i.e. automates the detection of *where* the leaking data of the target is stored in cache. In *S$A*, these two problems become independent. Hence, the attack is more challenging for the adversary, as the location problem needs to be solved before information can be extracted. However, since the extraction mechanism is the same, the *S$A* is applicable in all scenarios where *Flush+Reload* can be applied. We claim the *S$A* attack to be a substitute for the *Flush+Reload* attack whenever deduplication is not available. The added cost is the location step and a slightly decreased temporal resolution, since the (re-)priming needs to fill and check an entire set, not just a single line of cache. Hence, although this work demonstrates the applicability to AES only, the *S$A* attack is applicable in all cases where the *Flush+Reload* can be applied and has been applied. In other words, *S$A* can be applied to attack the public key cryptosystems targeted in [45, 43, 21]. This also means that focusing on countermeasures for AES is not helpful, since those will not prevent attacks on other crypto schemes also vulnerable to this attack.

6.2 AES-specific Countermeasures

Cache-based side channels are not a new phenomenon, hence numerous countermeasures have been proposed. The most obvious one is the use of AES-NI or other AES hardware extensions, if available on the processor. A good discussion of that and several other countermeasures like data independent memory accesses and smaller T-tables can be found in [38].

6.3 *S$A*-specific Countermeasures

Next, we discuss countermeasures that hinder the exploitability of the shared level cache and thereby prevent the *S$A* attack.

Dedicated servers: A simple solution to avoid cache side channel attacks is to utilize dedicated servers where the user's utilized virtual machines reside. Without the co-residency assumption *S$A*, *Flush+Reload* or any other trace driven attacks become unfeasible. Amazon EC2 offers to customers the posibility of using an isolated dedicated server for their computations.

Disable Huge Size Pages: In the particular case of the *S$A* cache side channel attack, if huge size pages are not allowed to be used by the guests the attack is no longer possible. The decision of using the huge size

pages could still be done *only* by the VMM, depending on certain parameters based on the length or the memory resources needed by the code.

Private L3 Cache Slices: One way to avoid the cache leakage that *S$A* uses is to make the cache slices private per VM, similar to the countermeasure suggested in [40]. This means that a particular VM is not allowed to interfere with the cache slice that another co-located VM is using. In this scenario the attacker does not interfere with the victim's cache slice and therefore cannot decide whether a specific memory line was used with *S$A*. This however, requires modifications to the cache arbitration mechanism and has the adverse affect of reducing the size of the cache slices made available to a single VM. It also limits the number of Guest VMs to the number of slices.

Hardware Masking of Addresses: Another possible solution is to apply a mask (implemented at the hardware level) to the offset field based on some of the non-set addressing bits in the physical address when huge size pages are used. Since the user no longer has control over the offset field, he cannot prime the specific set that he wants to target in the L3 cache and cannot decide whether the set was used or not by the victim.

Shadow Page Tables as Masking Option: In this case the shadow page tables that VMMs use for a virtual to virtual translation would play a more important role. For instance, the shadow page tables could not only handle the translation from VM virtual memory to VMM virtual memory, but also apply a mask based on the non cache-addressing bits. Thereby, the guest user does not know the masking value applied by the VM, and he cannot control the set that his data will occupy in the L3 cache.

7 Conclusion

***S$A*: A new deduplication free L3 cache side channel technique:** We proposed a new side channel technique that is applied in the L3 cache and therefore can be applied in cross-core scenarios. The new side channel technique bases its methodology in the usage of huge size pages, which give extra information about the position that each memory location occupies in the L3 cache.

Targeting a virtualized environemnt: We demonstrated that the new side channel technique can also be implemented in virtualized settings, particularly in Xen 4.1, where the usage of huge size pages by the guest OSs is allowed. Recall that the vast majority of the VMMs allow the usage of huge size pages, making *S$A* a suitable target for all of them.

Applying the attack on AES: We demonstrated the viability of the new side channel technique by recovering AES keys monitoring only 4 sets in the L3 cache in both virtualized and non-virtualized scenarios. In the noisier scenario the attack succeeds to recover the full AES key in less than 3 minutes. Thus, we showed that the efficiency of *S$A* is close to the efficiency achieved by *Flush+Reload* (which uses memory deduplication techniques) and is significantly higher than Bernstein's attack.

8 Disclosure

We have disclosed our attack to the security teams of VMware, Amazon AWS and Citrix.

References

1. Advanced Encryption Standard. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
2. Amazon Web Services. `http://aws.amazon.com/es/`.
3. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`.
4. How to Use Huge Pages to Improve Application Performance on Intel Xeon Phi Coprocessor . `https://software.intel.com/sites/default/files/Large_pages_mic_0.pdf`.

5. Huge Page Configuration in KVM. http://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaattunconfighp.htm?lang=en.
6. Intel Core i5-3320M Processor . http://ark.intel.com/products/64896/Intel-Core-i5-3320M-Processor-3M-Cache-up-to-3_30-GHz.
7. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.
8. Transparent Page Sharing: new default setting. http://blogs.vmware.com/security/2014/10/transparent-page-sharing-additional-management-capabilities-new-default-settings.html.
9. VMware Large Page performance . http://www.vmware.com/files/pdf/large_pg_performance.pdf.
10. X-XEN : Huge Page Support in Xen. https://www.kernel.org/doc/ols/2011/ols2011-gadre.pdf.
11. Xen 4.1 Release Notes. http://wiki.xen.org/wiki/Xen_4.1_Release_Notes.
12. CFS scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, April 2014.
13. Kernel Samepage Merging. http://kernelnewbies.org/Linux_2_6_32\#head-d3f32e41df508090810388a57efce73f52660ccb/, April 2014.
14. Aciiçmez, O. Yet Another MicroArchitectural Attack:: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 11–18.
15. Aciiçmez, O., Schindler, W., and Çetin K. Koç. Cache Based Remote Timing Attack on the AES. In *Topics in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007* (2007), Springer-Verlag, pp. 271–286.
16. Aciimez, O., Brumley, B. B., and Grabher, P. New Results on Instruction Cache Attacks. In *CHES* (2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 110–124.
17. Aly, H., and ElGayyar, M. Attacking AES Using Bernstein's Attack on Modern Processors. In *AFRICACRYPT* (2013), pp. 127–139.
18. Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the linux symposium* (2009), pp. 19–28.
19. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.
20. Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., and Butler, K. Detecting Co-residency with Active Traffic Analysis Techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2012), CCSW '12, ACM, pp. 1–12.
21. 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.
22. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.
23. Bonneau, J. Robust Final-Round Cache-Trace Attacks Against AES. *IACR Cryptology ePrint Archive 2006* (2006), 374.
24. 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/.
25. Cho, S., and Jin, L. Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 455–468.
26. Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 490–505.
27. Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
28. 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–.
29. 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/.
30. 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.
31. 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.
32. Mowery, K., Keelveedhi, S., and Shacham, H. Are AES x86 Cache Timing Attacks Still Feasible? In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2012), CCSW '12, ACM, pp. 19–24.
33. Neve, M., and Seifert, J.-P. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, E. Biham and A. Youssef, Eds., vol. 4356 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 147–162.

34. 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.
35. Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.
36. 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.
37. Spreitzer, R., and Plos, T. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security - NSS 2013, 7th International Conference, Madrid, Spain, June 3-4, 2013, Proceedings* (2013), R. S. Javier Lopez, Xinyi Huang, Ed., vol. 7873 of *Lecture Notes in COMPUTER Science*, Springer, pp. 656 – 662.
38. Tromer, E., Osvik, D., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.
39. 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.
40. Wang, Z., and Lee, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 494–505.
41. Weisberg, P., and Wiseman, Y. Using 4KB Page Size for Virtual Memory is Obsolete. In *Proceedings of the 10th IEEE International Conference on Information Reuse & Integration* (Piscataway, NJ, USA, 2009), IRI'09, IEEE Press, pp. 262–265.
42. Weiss, M., Heinz, B., and Stumpf, F. A Cache Timing Attack on AES in Virtualization Environments. In *14th International Conference on Financial Cryptography and Data Security (Financial Crypto 2012)* (2012), Lecture Notes in Computer Science, Springer.
43. 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.
44. 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.
45. 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.
46. Zhao, L., Iyer, R., Upton, M., and Newell, D. Towards Hybrid Last Level Caches for Chip-multiprocessors. *SIGARCH Comput. Archit. News 36*, 2 (May 2008), 56–63.