Wait a minute! A fast, Cross-VM attack on AES

Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar

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

Abstract. In cloud computing, efficiencies are reaped by resource sharing such as co-location of computation and deduplication of data. This work exploits resource sharing in virtualization software to build a powerful cache-based attack on AES. We demonstrate the vulnerability by mounting Cross-VM *Flush+Reload* cache attacks in VMware VMs to recover the AES keys of OpenSSL 1.0.1 running inside the victim VM. Furthermore, the attack works in a *realistic setting* where different VMs are located on separate cores. The modified *flush+reload* attack we present, takes only in the order of seconds to minutes to succeed in a cross-VM setting. Therefore long term co-location, as required by other fine grain attacks in the literature, are not needed. The results of this study show that there is a great security risk to OpenSSL AES implementation running on VMware cloud services when the deduplication is not disabled.

Keywords: Cross-VM, memory deduplication, flush+reload, cache attacks.

1 Introduction

In recent years we witnessed mass adoption of cloud based storage and compute systems such as Dropbox, Amazon EC2 and Microsoft Azure. Rather than acquiring and maintaining expensive workstations, clusters or servers, businesses can simply rent them from cloud service providers at the time of need. However, as with any new technology, cloud systems also come with problems of their own, namely co-residency data leakage problems. The data leakage problem is an indirect outcome of cloud's temperament. By definition a cloud system allows multiple users to share the same physical machine rather than assigning a dedicated machine to every user. Co-residency keeps the number of physical machines needed and the operating costs such as maintenance, electricity and cooling low but at a price. In cloud systems, different users run their virtual machines (VM) on the same physical machine separated only by a virtualization layer provided by a virtual machine manager (VMM) and supervised by a hypervisor. In theory *sandboxing* enforced by the VMM should suffice to completely isolate VMs from each other, but as elegantly stated many times: *"In theory there is no difference between theory and practice. But in practice, there is."*

A serious security problem that threatens VM isolation, stems from the fact that people are using software libraries that are designed to run on single-user servers and not on shared cloud hardwares and VM stacks. For privacy critical

data, especially cryptographic data, this gives rise to a blind spot where things may go wrong. Even though classical implementation attacks targeting cryptosystems featuring RSA and AES have been studied extensively, so far there has been little discussion about safe implementation of cryptosystems on cloud systems. For instance, implementation attacks on AES implementations, as proposed by Bernstein [8] and later [9, 13], use the timing difference of cache accesses to recover the secret key. A more recent study by Gullasch et.al [13] applies *Flush+Reload* attack between AES memory accesses. The attack recovers the key with as less as 100 encryptions. Even though these aforementioned methods have been implemented and the vulnerabilities are public, most cryptographic libraries still use vulnerable and unpatched implementations. Considering the level of access an adversary will have on a virtual machine, any of these attacks and many novel attacks can and will be realized on the cloud.

Another feature that can break process isolation or VM isolation is deduplication. Its exploitability has been shown in several studies. In 2011, Suzaki et al. [23] exploited an OS-optimization, namely Kernel Samepage Merging (KSM), to recover user data and subsequently identify a user from a co-located VM in a Linux Kernel-based Virtual Machine (KVM) [2] setting. In this study, authors were able to exploit the side-channel leakage to establish a covert communication channel between VMs and used this channel to detect co-residency with a target VM. Also in 2011 Suzaki et al. [22] exploited the same memory deduplication feature to detect processes like `sshd`, `apache2`, `IE6` and `Firefox` running on co-resident VM. The significance of this study is that not only it is possible to exploit the memory deduplication to detect the existence of a VM, but one can also detect the processes running on the target VM. This leads to cipher specific attacks and information thefts, as demonstrated by Suzaki et al. in [21]. In this latest study, the authors were able to detect security precautions such as antivirus software running on the co-resident target VM. Even though these studies paved the way for cross-VM process detection and shed light on vulnerabilities enabled by memory deduplication, a concrete attack recovering cryptographic keys has yet to be shown.

In [29] Weiß et al. for the first time presented a traditional cache timing attack on AES running inside a L4Re VM on an ARM Cortex-A8 single-core CPU with a `Fiasco.OC` microkernel. The attack is realized using Bernstein's correlation attack and targets several popular AES implementations including the one in `OpenSSL` [25]. The significance of this work is that it showed the possibility of extracting even finer grain information (AES vs. ElGamal keys in [32]) from a co-located VM. Recently, Irazoqui et al. [14] used Bernstein's attack to partially recover an AES key from a cross-VM attack running in XEN and VMware. While that work is the first one to show that fine-grain side-channel attacks can be mounted in cloud-like environments, the present attack is more efficient since it needs much less encryptions.

Our Contribution

In this work, we show a novel cache-based side-channel attack on AES that—by employing the *Flush+Reload* technique—enables, for the first time, a *practical* full key recovery attack across virtual machine boundaries in a *realistic* cloud-like server setting. The attack takes advantage of deduplication mechanism called the Transparent Page Sharing which is employed by VMware virtualization engine and is the focus of this work. The attack works well across cores, i.e. it works well in a high-end server with multiple cores scenario that is commonly found in cloud systems. The attack is, compared to [13], minimally invasive, significantly reducing requirements on the adversary: memory accesses are minimal and the accesses do not need to interrupt the victim process' execution. This also means that the attack is hardly detectable by the victim. Last but not least, the attack is lightning fast: we show that, when running in a realistic scenario where an encryption server is attacked, the whole key is recovered in less than 10 seconds in non-virtualized setting (i.e. using a spy process) even across cores, and in less than a minute in virtualized setting across VM boundaries.
In summary, this work

- shows for the first time that deduplication enables fine grain cross-VM attacks;
- introduces a new *Flush+Reload*-based attack that does not require interrupting the victim after each encryption round;
- presents the first *practical* cross-VM attack on AES; the attack is generic and can be adapted to any table-based block ciphers.

Since the presented attack is minimally invasive, it is very hard to detect. Finally, we also show that these attacks can be prevented without too much overhead.

After reviewing additional related work in Section 2 we detail on existing cache-based side-channel attacks in Section 3 and on memory deduplication in Section 4. The proposed attack is introduced in Section 5. Results are presented in Section 6. Before concluding in Section 8 we discuss possible countermeasures in Section 7.

2 Related Work

Over the last decade, a great number of research has been done in the field of cache-based side-channel attacks. The first time driven attack was done by Bernstein when he observed that non-constant time implementations of cryptographic algorithms leak sensitive information in terms of time which can be used to extract the secret key [8]. His target was the OpenSSL implementation of the cryptographic algorithm AES. Even though that the side-channel attack that he described works, he did not give an explanation as why. Neve [17], in his PhD thesis explained that Bernstein's attack exploits the access time differences of different cache lines.

Bonneau and Mironov's study [9] shows how to exploit cache collisions in AES as a source for time leakage. In the same line, Acıiçmez and Koç [4] investigated a collision timing attack in the first and the second round of AES. More trace driven attacks were investigated by Osvik et al. [19] where they tried the *prime and probe* attack on AES. In the aforementioned study, a *spy process* fills the cache with attacker's own data and then waits for the victim to run the encryption. When the encryption is finished, the attacker tries to access her own data and measures the access time to see which cache lines have been evicted from the cache. Then, comparing the access times with the reference ones, attacker discovers which cache lines were used. In the same study, authors also analyze *evict+time* method that consists of triggering two encryptions of the same plaintext and accessing some cache lines after the first encryption to see which lines are again loaded by the second encryption. Also, in another study done by Gullasch et al. [13] *flush+reload* is used to attack AES encryption by blocking the execution of AES after each memory access.

Even though AES is a popular target for side-channel cache attacks, it is not the only target. O.Acıiçmez in [5] was the first one discovering that the instruction cache as well as the data cache leaked information when performing RSA encryption. Brumley and Boneh performed a practical attack against RSA in [10]. Later Chen et al. developed the trace driven instruction cache attacks on RSA. Finally Yarom et al. were the first ones proposing a *flush+reload* attack on RSA using the instruction cache [31]. Finally, again Yarom et al. used the *Flush+Reload* technique to recover the secret key from a ECDSA signature algorithm [30].

In a cloud environment, several studies have been conducted with the aim of breaking the isolation between co-located VMs to perform side-channel attacks. In 2012, Ristenpart et al. [20] demonstrated that it is possible to solve the co-location problem in the cloud environment and extract sensitive data from a targeted VM. In the study, Amazon's EC2 servers were targeted and using their IP addresses provided by Amazon, VMs were mapped to various types of cloud instances. Using a large set of IP-instance type matches and some network delay timing measurements, they were able to identify where a particular target VM is likely to reside, and then instantiate new VMs until one becomes co-resident with the target VM. Along with the placement information, they exploited Amazon EC2's sequential placement policy and were able to co-locate two VMs on a single physical machine with 8% probability. Even further, the authors show how cache contention between co-located Xen VMs may be exploited to deduce keystrokes with high success probability. By solving the co-location problem, this initial result fueled further research in Cross-VM side-channel attacks.

After solving the co-location problem, stealing fine grain secret information from a target turns into an ordinary side-channel cache attack. In 2012, Zhang et al. [32] presented an access-driven side-channel attack implemented across Xen VMs that manages to extract fine-grain information from a victim VM. In the study, authors managed to recover an ElGamal decryption key from a victim VM using a cache timing attack. The significance of this work, is that

for the first time the authors were able to extract *fine grain* information across VMs—in contrast to the earlier work of Ristenpart et al. [20] who managed to extract keystroke patterns. Later, Yarom et al. in [31] suggested that their attack could be used in a virtualized environment but they never tried it in a real cloud environment. Again, for the AES case, Weiss et al. used Bernstein's attack on an ARM system in a virtualized environment to extract information about AES encryption keys [29].

Finally in 2014 Irazoqui et al. [14] implemented Bernstein's attack for the first time in a virtualized environment where Xen and VMware VMMs with cross-VM setting were used. In the study, authors were able to recover AES secret key from co-resident VM running AES encryption using the timing difference between cache line accesses. The downside of the attack was that average of 2^{29} encryption samples were needed for the attack to work which takes about 4-5 hours on a modern Core i5 platform.

3 Cache-Based Side-Channel Attacks

In this work we demonstrate a fine-grain cross-VM attack that one might use in the real world. We not only want the attack to allow us to recover fine-grain information, but also work in a reasonable amount of time, with assumptions one can fulfill rather easily on cloud systems. Since Bernstein's attack [8] numerous trace-driven, access-driven and time-driven attacks have been introduced mainly targeting AES implementations. We will employ a new variant: **the flush and reload attack** on AES. In what follows we explain the basics of cache side-channel attacks, and briefly review the many cache side-channel attacks that have been used to attack AES.

Cache Architecture. The cache architecture consists of a hierarchy of memory components located between the CPU cores and the RAM. The purpose of the cache is to reduce the average access time to the main memory by exploiting locality principles. When the CPU needs to fetch data from memory, it queries the cache memory first to check if the data is in the cache. If it is, then it can be accessed with much smaller delay and in this case it is said that a cache *hit* has occurred. When the data is not present in the cache, it needs to be fetched from a higher-level cache or even from main memory. This results in greater delays. This case is referred to as a cache *miss*. When a cache miss occurs, the CPU retrieves the data from the memory and a copy is stored in the cache. The CPU loads bigger blocks of data, including data in nearby locations, to take advantage of *spatial locality*. Loading the whole block of data improves the execution performance because values stored in nearby locations to the originally accessed data are likely to be accessed.

The cache is organized into fixed sized *cache lines*, e.g of l bytes each. A cache line represents the partitions of the data that can be retrieved or written at a time when accessing the cache. When an entry of a table stored in memory is accessed for the first time, the memory line containing the retrieved data is loaded into the cache. If the process tries to access to the same data from

the same memory line again, the access time will be significantly lower, i.e. a cache hit occurs. Therefore—for a cryptographic process—the encryption *time* depends directly on the accessed table positions, which in turn depend on the secret internal state of the cipher. This timing information can be exploited to gain information about the secret key that is being used in the encryption. Also, in case that there are no empty (invalid) cache lines available, one of the data bearing lines gets reallocated to open up space for the the incoming line. Therefore, cache lines that are not recently accessed are *evicted* from cache.

Exploiting Cache Timing Information. Up until this point, we established that a cache miss takes more time to be processed than a cache hit. Using the resulting *state-dependent* timing information, an attacker can obtain sensitive information from an encryption process and use this information to recover information about the secret key, eventually resulting in a full key recovery. The run-time of a fast software implementation of a cipher like AES [18] often heavily depends on the speed at which table look ups are performed. A popular implementation style for the AES is the T table implementation of AES [11] which combines the `SubBytes`, `ShiftRows` and `MixColumns` operations into one single table look up per state byte, along with XOR operations. This operation is called the *TableLookUp* operation. The advantage of this implementation style is that it allows the computation of one round using only table look-ups and XOR operations which is much faster than performing the actual finite-field arithmetic and logic operations. Compared to using standard S-boxes, T table based implementations use more memory, but the encryption time is significantly reduced, especially on 32-bit CPUs. For this reason, *almost all* of the current software implementations of the AES encryption for high-performance CPUs are T table implementations.

Note that the index of the loaded table entry is determined by a byte of the cipher state. Hence, information on which table values have been loaded into cache can reveal information about the secret state of AES. Such information can be retrieved by monitoring the cache directly, as done in *trace-driven* cache attacks. Similar information can also be learned by observing the timing behavior of multiple AES executions over time, as done in *time-driven* cache attacks. Finally, there are *access driven* cache attacks, which require the attacker to learn which cache lines have been accessed (like trace-driven attacks), but (like timing-driven attacks) do not require detailed knowledge on when and in what order the data was accessed. So the difference between these classes of attacks is the attacker's access capabilities:

- **Time driven attacks** are the least restrictive type with the only assumption that the attacker can observe the aggregated timing profile of a full execution of a target cipher.
- **Trace driven** attacks assume the attacker has access to the cache profile when the targeted program is running.
- **Access driven** attacks assume only to know which sets of the cache have been accessed during the execution of a program.

The attacks presented in this paper belong to a sub-class of *access-driven* cache attacks, which we discuss next.

3.1 The Flush+Reload Technique

The *Flush+Reload* attack is a powerful cache-based side-channel attack technique first proposed in [13], but was first named in [31]. It can be classified as an access driven cache attack. It usually employs a spy process to ascertain if specific cache lines have been accessed or not by the code under attack. Gullasch et al. [13] first used this spy process on AES, although the authors did not brand their attack as *Flush+Reload* at the time. Here we briefly explain how *Flush+Reload* works. The attack is carried out by a spy process which works in 3 stages:

Flushing stage: In this stage, the attacker uses the `clflush` command to flush the desired memory lines from the cache hence make sure that they have to be retrieved from the main memory next time they need to be accessed. We have to remark here that the `clflush` command does not only flush the memory line from the cache hierarchy of the corresponding working core, but it flushes from all the caches of all the cores in the PC. This is an important point: if it only flushed the corresponding core's caches, the attack would only work if the attacker and victim's processes were co-residing on the same core. This would have required a much stronger assumption than just being in the same physical machine.

Target accessing stage: In this stage the attacker waits until the target runs a fragment of code, which might use the memory lines that have been flushed in the first stage.

Reloading stage: In this stage the attacker reloads again the previously flushed memory lines and measures the time it takes to reload. Depending on the reloading time, the attacker decides whether the victim accessed the memory line in which case the memory line would be present in the cache or if the victim did not access the corresponding memory line in which case the memory line will not be present in the cache. The timing difference between a cache hit and a cache miss makes the aforementioned access easily detectable by the attacker.

The fact that the attacker and the victim processes do not reside on the same core is not a problem for the *Flush+Reload* attack because even though there can exist some isolation at various levels of the cache, in most systems there is some level shared between all the cores present in the physical machine. Therefore, through this shared level of cache (typically the L3 cache), one can still distinguish between accesses to the main memory.

4 Memory Deduplication

Memory deduplication is an optimization technique that was originally introduced to improve the memory utilization of VMMs. It later found its way into

common non-virtualized OSs as well. Deduplication works by recognizing processes (or VMs) that place the same data in memory. This frequently happens when two processes use the same shared libraries. The deduplication feature eliminates multiple copies from memory and allows the data to be shared between users and processes. This method is especially effective in virtual machine environments where multiple guest OSs co-reside on the same physical machine and share the physical memory. Consequently, variations of memory deduplication technology are now implemented in both the VMware [26, 27] and the KVM [3, 16] VMMs. Since KVM converts linux kernel into a hypervisor, it directly uses KSM as page sharing technique, whereas VMware uses what is called Transparent Page Sharing(TPS). Although they have different names, their mechanism is very similar; the hypervisor looks for identical pages between VMs and when it finds a collision, it merges them into one single page.

Even though the deduplication optimization method saves memory and thus allows more virtual machines to run on the host system, it also opens door to side-channel attacks. While the data in the cache cannot be modified or corrupted by an adversary, parallel access rights can be exploited to reveal secret information about processes executing in the target VM. Also, an adversary can *prime* the cache and wait for the victim to access some of this primed data. The accessed/replaced cache data reveals information about the victims behavior. In this study, we will focus on the Linux implementation of Kernel Samepage Merging (KSM) memory deduplication feature and on TPS mechanism implemented by VMware.

4.1 KSM (Kernel Same-page Merging)

KSM is the Linux memory deduplication feature implementation that first appeared in Linux kernel version 2.6.32 [16, 3]. In this implementation, KSM kernel daemon `ksmd`, scans the user memory for potential pages to be shared among users [7]. Also, since it would be CPU intensive and time consuming, instead of scanning the whole memory continuously, KSM scans only the potential candidates and creates signatures for these pages. These signatures are kept in the deduplication table. When two or more pages with the same signature are found, they are cross-checked completely to determine if they are identical. To create signatures, the KSM scans the memory at 20 msec intervals and at best only scans the 25% of the potential memory pages at a time. This is why any memory disclosure attack, including ours, has to wait for a certain time before the deduplication takes effect upon which the attack can be performed. During the memory search, the KSM analyzes three types of memory pages [24];

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

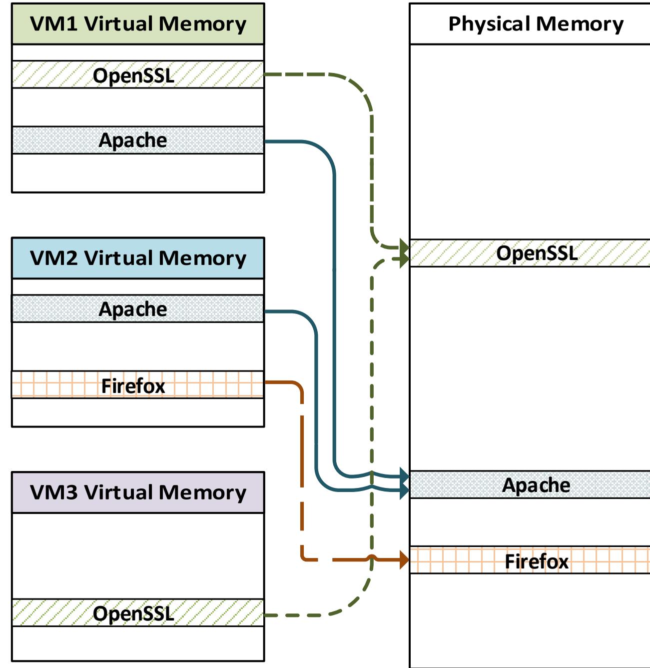

Fig. 1. Memory Deduplication Feature

When a duplicate page signature is found among candidates and the contents are cross-checked, `ksmd` automatically tags one of the duplicate pages with copy-on-write (COW) tag and shares it between the processes/users while the other copy is eliminated. Experimental implementations [3] show that using this method, it is possible to run over 50 Windows XP VMs with 1GB of RAM each on a physical machine with just 16GB of RAM. As a result of this, the power consumption and system cost is significantly reduced for systems with multiple users.

5 CFS-free Flush+Reload Attack on AES

In this section we will describe the principles of our *Flush+Reload* attack on the AES implementation of `OpenSSL`. In [13] Gullasch et al. described a *Flush+Reload* attack on AES implementation of the `OpenSSL` library. However in this study, we are going to use the *Flush+Reload* method with some modifications that from our point of view, have clear advantages over [13]. Prior to the comparison with other cache side channel attacks, a detailed explanation of our *Flush+Reload* spy process is given along with the attack steps. We consider two scenarios: the attack as a spy process running in the same OS instance as the victim (as done

in [13]), and the attack running as a cross-VM attack in a virtualized environment.

5.1 Description of the Attack

As in prior *Flush+Reload* attacks, we assume that the adversary can monitor accesses to a given cache line. However, unlike the attack in [13], this attack

- only requires the monitoring of a *single* memory line; and
- flushing can be done before encryption, reloading after encryption, i.e. the adversary does not need to interfere with or interrupt the attacked process.

More concretely, the Linux kernel features a completely fair scheduler which tries to evenly distribute CPU time to processes. Gullasch et al. [13] exploited Completely Fair Scheduler (CFS) [15, 1], by overloading the CPU while a victim AES encryption process is running. They managed to gain control over the CPU and suspend the AES process thereby gaining an opportunity to monitor cache accesses of the victim process. Our attack is agnostic to CFS and does not require time consuming overloading steps to gain access to the cache.

We assume the adversary monitors accesses to a single line of one of the T tables of an AES implementation, preferably a T table that is used in the last round of AES. Without loss of generality, let's assume the adversary monitors the memory line corresponding to the first positions of table T, where T is the lookup table applied to the targeted state byte s_i, where s_i is the i-th byte of the AES state before the last round. Let's also assume that a memory line can hold n T table values, e.g, the first n T table positions for our case. If s_i is equal to one of the indices of the monitored T table entries in the memory line (i.e. $s_i \in \{0, \dots, n\}$ if the memory line contains the first n T table entries) then the monitored memory line will with very high probability be present in the cache (since it has been accessed by the encryption process). However, if s_i takes different values, the monitored memory line is not loaded in this step. Nevertheless, since each T table is accessed l times (for AES-128 in `OpenSSL`, $l = 40$ per T_j), there is still a probability that the memory line was loaded by any of the other accesses. In both cases, all that happens after the T table lookup is a possible reordering of bytes (due to AES's `Shift_Rows`), followed by the last round key addition. Since the last round key is always the same for s_i, the n values are mapped to n specific and constant ciphertext byte values. This means that for n out of 256 ciphertext values, the monitored memory line will *always* have been loaded by the AES operation, while for the remaining $256 - n$ values the probability of having been reloaded is smaller. In fact, the probability that the specific T table memory line i has not been accessed by the encryption process is given as:

$$\Pr\left[\text{no access to } \ T[i]\right] = \left(1 - \frac{t}{256}\right)^{l}$$

Here, l is the number of accesses to the specific T table. For `OpenSSL 1.0.1` AES-128 we have $l = 40$. If we assume that each memory line can hold $t = 8$

Algorithm 1: Recovery algorithm for key byte k_0

```
Input  : X_0                  //Reload vector for ciphertext byte 0
Output: k_0                   //Correct key byte 0
forall x_j ∈ X_0 do
    //Threshold for values with low reload counter.
    if x_j < Low_counter_threshold then
        for s = 0 to n do
            //xor with each value of the targeted T table memory line
            K_0[j ⊕ T[s]]++;
        end
    end
end
return argmax_k(K_0[k]);
```

entries per cache line, we have Pr [no access to $T[i]] = 28\%$. Therefore it is easily distinguishable whether the memory line is accessed or not. Indeed, this turns out to be the case as confirmed by our experiments.

In order to distinguish the two cases, all that is necessary is to measure the timing for the *reload* of the targeted memory line. If the line was accessed by the AES encryption, the reload is quick; else it takes more time. Based on a threshold that we will empirically choose from our measurements, we expect to distinguish main memory accesses from L3 cache accesses. For each possible value of the ciphertext byte c_i we count how often either case occurs. Now, for n ciphertext values (the ones corresponding to the monitored T table memory line) the memory line has always been reloaded by AES, i.e. the reload counter is (close to) zero. These n ciphertext values are related to the state as follows:

$$c_i = k_i \oplus T\left[s_{[i]}\right] \tag{1}$$

where the $s_{[i]}$ can take n consecutive values. Note that Eq. (1) describes the last round of AES. The brackets in the index of the state byte $s_{[i]}$ indicate the reordering due to the `Shift_Rows` operation. For the other values of c_i, the reload counter is significantly higher. Given the n values of c_i with a low reload counter, we can solve Eq. (1) for the key byte k_i, since the indices $s_{[i]}$ as well as the table output values $T\left[s_{[i]}\right]$ are known for the monitored memory line. In fact, we get n possible key candidates for each c_i with a zero reload counter. The correct key is the only one that all n valid values for c_i have in common.

A general description of the key recovery algorithm is given in Algorithm 1, where key byte number 0 is recovered from the ciphertext values corresponding to n low reload counter values that were recovered from the measurements. Again, n is the number of T table positions that a memory line holds. The *reload vector* $X_i = [x(0), x(1), \ldots, x(255)]$ holds the reload counter values $x(j)$ for each ciphertext value $c_i = j$. Finally K_0 is the vector that, for each key byte candidate k, tracks the number of appearances in the key recovery step.

Example Assume that the memory line can hold $n = 4$ T table values and we want to recover key byte k_0. There are four ciphertext values detected with a low reload counter. Assume further that each c_0 has been xored with the T table values of the monitored memory line (the first 4 if we are working with the first positions), giving $k_0^{(i)} = c_0^i \oplus T\left[s_{[0]}\right]$. For each of the four possibilities of c_0, there are $n = 4$ possible solutions for k_0. If the results are the following:

$$k_0^{(0)} \begin{cases} 43 \\ ba \\ 91 \\ 17 \end{cases} k_0^{(1)} \begin{cases} 8b \\ 91 \\ f3 \\ 66 \end{cases} k_0^{(2)} \begin{cases} 91 \\ 45 \\ 22 \\ af \end{cases} k_0^{(3)} \begin{cases} cd \\ 02 \\ 51 \\ 91 \end{cases}$$

And since there is only one common solution between all of them, which is 91, we deduce that the correct key value is $k_0 = 91$. This also means that $K_0[91] = 4$, since $k = 91$ appeared four times as possible key candidate in the key recovery step.

Note that this is a *generic attack* that would apply virtually to any table-based block cipher implementation. That is, our attack can easily be adapted to other block ciphers as long as their last round consists of a table look-up with a subsequent key addition.

5.2 Recovering the Full Key

To recover the full key, the attack is expanded to all tables used in the last round, e.g. the 4 T tables of AES in `OpenSSL 1.0.1`. For each ciphertext byte it is known which T table is used in the final round of the encryption. This means that the above attack can be repeated on each byte, by simply analyzing the collecting ciphertexts and their timings for each of the ciphertext bytes individually. As before, the timings are profiled according to the value that each ciphertext byte c_i takes in each of the encryptions, and are stored in a ciphertext byte vector. The attack process is described in Algorithm 2. In a nutshell, the algorithm monitors the first T table memory line of all used tables and hence stores four reload values per observed ciphertext. Note that, this is a *known ciphertext attack* and therefore all that is needed is a flush of one memory line before one encryption. There is no need for the attacker to gain access to plaintexts.

Finally the attacker should apply Algorithm 1 to each of the obtained ciphertext reload vectors. Recall that each ciphertext reload vector uses a different T table, so the right corresponding T table should be applied in the key recovery algorithm.

Performing the Attack. In the following we provide the details about the process followed during the attack.

Step 1: Acquire information about the offset of T tables The attacker has to know the offset of the T tables with respect to the beginning of the library. With that information, the attacker can refer and point to any memory line that holds T table values even when the ASLR is activated. This means that

Algorithm 2: `Flush and reload` algorithm extended to 16 ciphertext bytes

```
Input  : T0_0, T1_0, T2_0, T3_0              //Addresses of each T table
Output: X_0, X_1, ...X_15                    //Reload vectors for ciphertext bytes
                                             //Each X_k holds 256 counter values
while iteration < total number of measurements do
    clflush(T0_0, T1_0, T2_0, T3_0);         //Flush data to the main memory
    ciphertext=Encryption(plaintext);        //No need to store plaintext!
    for i ← T0_0 to T3_0 do
        time=Reload(i);
        if time > AccessThreshold then
            Addcounter(T_i,X_i);             //Increase counter of X_i using T_i
        end
    end
end
return X_0, X_1, ..., X_15
```

some reverse engineering work has to be done prior to the attack. This can be done in a debugging step where the offset of the addresses of the four T tables are recovered.

Step 2: Collect Measurements In this step, the attacker requests encryptions and applies *Flush+Reload* between each encryption. The information gained, i.e. Ti_0 was accessed or not, is stored together with the observed ciphertext. The attacker needs to observe several encryptions to get rid of the noise and to be able to recover the key. Note that, while the reload step must be performed and timed by the attacker, the flush might be performed by other processes running in the victim OS.

Step 3: Key recovery In this final step, the attacker uses the collected measurements and his knowledge about the public T tables to recover the key. From this information, the attacker applies the steps detailed in Section 5.1 to recover the individual bytes of the key.

5.3 Attack Scenario 1: Spy Process

In this first scenario we will attack an encryption server running in the same OS as the spy process. The encryption server just receives encryption requests, encrypts a plaintext and sends the ciphertext back to the client. The server and the client are running on different cores. Thus, the attack consists in distinguishing accesses from the last level of cache, i.e. L3 cache, which is shared across cores. and the main memory. Clearly, if the attacker is able to distinguish accesses between last level of cache and main memory, it will be able to distinguish between L1 and main memory accesses whenever server and client co-reside in the same core. In this scenario, both the attacker and victim are using the same shared library. The KSM is responsible for merging those pages into one unified shared

page. Therefore, the victim and attacker processes are linked through the KSM deduplication feature.

Our attack works as described in the previous section. First the attacker discovers the offset of the addresses of the T tables with respect to the begining of the library. Next, it issues encryption requests to the server, and receives the corresponding ciphertext. After each encryption, the attacker checks with the *Flush+Reload* technique whether the chosen T table values have been accessed. Once enough measurements have been acquired, the key recovery step is performed. As we will see in our results section, the whole process takes less than half a minute.

Our attack significantly improves on previous cache side-channel attacks such as *evict + time* or *prime and probe* [19]. Both attacks were based on spy processes targeting the L1 cache. A clear advantage of our attack is that —since it is targeting the last shared level cache— it works across cores. Of course both *evict + time* or *prime and probe* attacks can be applied to the last level of cache, but their performance would be significantly reduced in cross-core setting, due to the large number of evictions/probings that are needed for a successful attack.

A more realistic attack scenario was proposed earlier by Bernstein [8] where the attacker targets an encryption server. Our attack similarly works under a realistic scenario. However. unlike Bernstein's attack [8], our attack does not require a profiling phase that involves access to an identical implementation with a known-key. Finally, with respect to the previous *Flush+Reload* attack in AES, our attack does not need to interrupt the AES execution of the encryption server. We will compare different attacks according to the number of encryptions needed in Section 6.1.

5.4 Attack Scenario 2: Cross-VM Attack

In our second scenario the victim process is running in one virtual machine and the attacker in another one but on the same machine possibly on different cores. For the purposes of this study it is assumed that the co-location problem has been solved using the methods proposed in [20], ensuring the attacker and the victim are running on the same physical machine. The attack exploits memory overcommitment features that some VMMs such as VMware provide. In particular, we focus in memory deduplication. The VMM will search periodically for identical pages across VMs to merge both pages into a single page in the memory. Once this is done (without the intervention of the attacker) both the victim and the attacker will access the same portion of the physical memory enabling the attack. The attack process is the same as in Scenario 1. Moreover, we later show that the key is recovered in less than a minute, which makes the attack quite *practical*.

We discussed the improvements of our attack over previous proposals in the previous scenario except the most important one: We believe that the `evict+time`, `prime and probe` and time collision attacks will be rather difficult to carry out in real cloud environment. The first two are targeting the L1 cache, which is not

shared across cores. The attacker would have to be in the same core as the victim, which is a much stronger assumption than being just in the same physical machine. Both `evict+time` and `prime and probe` could be applied to work with the L3 cache, but the noise and the amount of measurements would need to be drastically increased. Even further, due the increasing amount of source noises present in a cloud scenario (more layers, network latency) both `evict+time` and time collision attacks would be hard to perform. Finally, targeting the CFS [13] to evict the victim process, requires for the attacker's code to run in the same OS, which will certainly not be possible in a virtualized environment.

6 Experiment Setup and Results

We present results for both a spy process within the native machine as well as the cross-VM scenario. The target process is executed in Ubuntu 12.04 64 bits, kernel version 3.4, using the AES implementation of `OpenSSL 1.0.1f` for encryption. All experiments were performed on a machine featuring an Intel i5-3320M four core clocked at 3.2GHz. The Core i5 has a three-level cache architecture: The L1 cache is 8-way associative, with 2^{15} bytes of size and a cache line size of 64 bytes. The level-2 cache is 8-way associative as well, with a cache line width of 64 bytes and a total size of 2^{18} bytes. The level-3 cache is 12-way associative with a total size of 2^{22} bytes and 64 bytes cache line size. It is important to note that each core has private L1 and L2 caches, but the L3 cache is shared among all cores. Together with the deduplication performed by the VMM, the shared L3 cache allows the adversary to learn about data accesses by the victim process.

The *attack scenario* is as follows: the victim process is an encryption server handling encryption requests through a socket connection and sends back the ciphertext, similar to Bernstein's setup in [8]. But unlike Bernstein's attack, where packages of at least 400 bytes were sent to deal with the noise, our server only receives packages of 16 bytes (the plaintext). The encryption key used by the the server is unknown to the attacker. The attack process sends encryption queries to the victim process. All measurements such as timing measurements of the reload step are done on the attacker side. The server uses `OpenSSL 1.0.1f` for the AES encryption. In our setup, each cache line holds 16 T table values, which results in a 7.6% probability for not accessing a memory line per encryption. All given attack results target only the first cache line of each T table, i.e. the first 16 values of each T table for flush and reload. Note that in the attack any memory line of the T table would work equally well. Both native and cross-VM attacks establish the threshold for selecting the correct ciphertext candidates for the working T table line by selecting those values which are below half of the average of overall timings for each ciphertext value. This is an empirical threshold that we set up after running some experiments as follows

$$\text{threshold} = \sum_{i=0}^{256} \frac{t_i}{2 \cdot 256} \ .$$

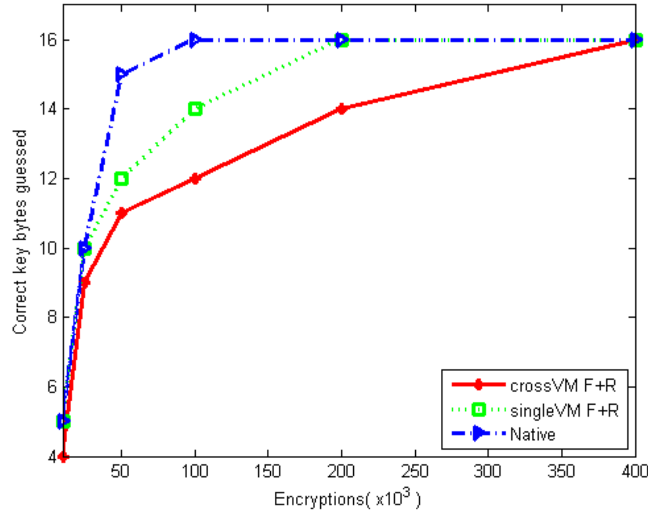

Fig. 2. Number of correct key bytes guessed of the AES-128 bit key vs. number of encryption requests. Even 50.000 encryptions (i.e. less than 5 seconds of interaction) result in significant security degradation in both the native machine as well as the cross-VM attack scenario.

Spy Process Attack Setup: The attack process runs in the same OS as the victim process. The communication between the processes is carried out via localhost connection and measures timing using Read Time-Stamp Counters (`rdtsc`). The attack is set up to work across cores; the encryption server is running in a different core than the attacker. We believe that distinguishing between L3 and main memory accesses will be more susceptible to noise than distinguishing between L1 cache accesses and main memory accesses. Therefore while working with the L3 cache gives us a more realistic setting, it also makes the attack more challenging.

Cross-VM Attack Setup: In this attack we use VMware ESXI 5.5.0 build number 1623387 running Ubuntu 12.04 64-bits guest OSs. We know that VMware implements TPS with large pages (2 MB) or small pages (4 KB). We decided to use the later one, since it seems to be the default for most systems. Furthermore, as stated in [26], even if the large page sharing is selected, the VMM will still look for identical small pages to share. For the attack we used two virtual machines, one for the victim and one for the attacker. The communication between them is carried out over the local IP connection.

The results are presented in Figure 2 which plots the number of correctly recovered key bytes over the number of timed encryptions. The dash-dotted line shows that the spy-process scenario completely recovers the key after only 2^{17} encryptions. Prior to moving to the cross-VM scenario, a single VM scenario was

performed to gauge the impact of using VMs. The dotted line shows that due to the noise introduced by virtualization we need to nearly double the number of encryptions to match the key recovery performance of the native case. The solid line gives the result for the cross-VM attack: 2^{19} observations are sufficient for stable full key recovery. The difference might be due to cpuid like instructions which are emulated by the hipervisor, therefore introducing more noise to the attack. In the worst case, both the native spy process and the single VM attack took around 25 seconds (for 400.000 encryptions). We believe that this is due to communication via the localhost connection. However when we perform a cross-VM attack it takes roughly twice as much time as in the previous cases. In this case we are performing the communication via local IPs that have to reach the router, which is believed to add the additional delay. This means that *all of the described attacks —even in the cross VM scenario— completely recover the key in less than one minute*!

6.1 Comparison to Other Attacks

Next we compare the most commonly implemented cache-based side-channel attacks to the proposed attack. Results are shown in Table 1. It is difficult to compare the attacks, since most of them have been run on different platforms. Many of the prior attacks target OpenSSL's 0.9.8 version of AES. Most of these attacks exploit the fact that AES has a separate T Table for the last round, significantly reducing the noise introduced by cache miss accesses. Hence, attacks on OpenSSL0.9.8's AES usually succeed much faster, a trend confirmed by our attack results. Note that our attack, together with [6] and [14] are the only ones that have been run on a 64 bit processor. Moreover, we assume that due to undocumented internal states and advanced features such as hardware prefetchers, implementation on a 64 bit processor will add more noise than older platforms running the attack. With respect to the number of encryptions, we observe that the proposed attack has significant improvements over most of the previous attacks.

Spy process in native OS: Even though our attack runs in a noisier environment than Bernstein's attack, *evict and time*, and cache timing collision attacks, it shows better performance. Only *prime and probe* and *Flush+Reload* using CFS show either comparable or better performance. The proposed attack has better performance than *prime and probe* even though their measurements were performed with the attack and the encryption being run as one unique process. The *Flush+Reload* attack in [13] exploits a much stronger leakage, which requires that attacker **to interrupt the target AES between rounds** (an unrealistic assumption). Furthermore, *Flush+Reload* with CFS needs to monitor the entire T tables, while our attack only needs to monitor a single line of the cache, making the attack much more lightweight and subtle.

Cross-VM attack: So far there is only one publication that has analyzed cache-based leakage across VMs for AES [14]. Our proposed attack shows dramatic improvements over [14], which needs 2^{29} encryptions (hours of run time) for a

Table 1. Comparison of cache side-channel attack techniques against AES.

Attack	Platform	Methodology	OpenSSL	Traces
Spy-Process based Attacks:				
Collision timing [9]	Pentium 4E	Time measurement	`0.9.8a`	300.000
Prime+probe [19]	Pentium 4E	L1 cache prime-probing	`0.9.8a`	16.000
Evict+time [19]	Athlon 64	L1 cache evicting	`0.9.8a`	500.000
Flush+reload (CFS) [13]	Pentium M	*Flush+reload* w/CFS	`0.9.8m`	100
Our attack	i5-3320M	L3 cache *Flush+reload*	`0.9.8a`	8.000
Bernstein [6]	Core2Duo	Time measurement	`1.0.1c`	2^{22}
Our attack	i5-3320M	L3 cache *Flush+reload*	`1.0.1f`	100.000
Cross-VM Attacks:				
Bernstein [14][1]	i5-3320M	Time measurement	`1.0.1f`	2^{30}
Our attack(VMware)	i5-3320M	L3 cache *Flush+reload*	`1.0.1f`[2]	400.000

[1] Only parts of the key were recovered, not the whole key.
[2] The AES implementation was not updated for the recently released OpenSSL 1.0.1g and 1.0.2 beta versions. So the results for those libraries are identical.

partial recovery of the key. Our attack only needs 2^{19} encryptions to recover the full key. Thus, while the attack presented in [14] needs to interact with the target for several hours, our attack succeeds in under a minute and recovers the entire key. Note that, the CFS enabled *Flush+Reload* attack in [13] will not work in the cross-VM setting, since the attacker has no control over victim OS's CFS.

7 Countermeasures

AES-NI: Using AES-NI instructions solves the cache-access leakage for AES. In this case the AES encryption does not use the memory but it uses specific hardware instructions, avoiding the possibility of implementing a cache-based side-channel attack completely. However, AES is not the only symmetric cipher in use nowadays: the problem remains for other encryption algorithms for which hardware acceleration is not provided.

Cache Flushing: Flushing each of the T table values after the AES execution will have the similar consequences as prefetching them before the execution [14]. When the attacker wants to decide whether a line has been accessed, he will find that the T tables are in the memory and therefore, he will not see any time differences. Again this implies a higher execution time. With such a countermeasure the only possibility left to the attacker is to block the AES execution during some of its rounds (as done in [13]). Hence, this would mitigate cross-VM attacks and require a more advanced attacker than we considered for our attack.

Restricting the Deduplication: Disabling the deduplication would make the attack impossible in the cloud however memory deduplication is highly performance beneficial, especially in cloud where multiple users share the same hard-

ware. This is why we believe that the system designers should restrict the deduplication mechanism rather then completely disabling it. Since the `madvise` [12] system call that manages the deduplication process, does not scan all of the memory pages but scans only the selected portions of the memory, one can exploit this feature and limit the resource sharing between VMs. This limitation can either be on hardware or software level. As suggested by Wang and Lee [28] the OS can enforce a smart process scheduling method to protect critical processes with sensitive data and make sure that they are never shared between VMs.

8 Conclusion

***Flush+Reload* in AES: A New Fine Grain Attack:** Our experiments show that if applied in a clever way, *Flush+Reload* is a fine grain attack on AES and can recover the whole key. Furthermore, the attack can be applied to any block cipher that uses a T table based implementation. The attack has to take advantage of deduplication so that victim and attacker share the same memory.

Making The Attack Feasible in The Cloud: We not only performed the attack in native machine, but also in a cloud-like cross-VM scenario. Although there is more noise in the latter scenario, the attack recovers the key with just 400.000 encryptions. In this case, the attacker has to take advantage of some memory sharing mechanism (such as TPS in VMware).

Lightning-Fast Attack: Even in the worst case scenario (cross-VM) the attack succeeds in less than a minute. To the best of our knowledge, no faster attack has been implemented against AES in a *realistic* cloud-like setting. This also means that just one minute of co-location with the encryption server suffices to recover the key.

9 Acknowledgments

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

References

1. Cfs scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, April 2014.
2. Kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`, April 2014.
3. Kernel samepage merging. `http://kernelnewbies.org/Linux_2_6_32\#head-d3f32e41df508090810388a57efce73f52660ccb/`, April 2014.
4. Aciiçmez, O., and Koç, Ç. K. Trace-driven cache attacks on aes (short paper). In *Information and Communications Security*. Springer, 2006, pp. 112–121.

5. Aciimez, 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.
6. Aly, H., and ElGayyar, M. Attacking aes using bernstein's attack on modern processors. In *AFRICACRYPT* (2013), pp. 127–139.
7. Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using ksm. In *Proceedings of the linux symposium* (2009), pp. 19–28.
8. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.
9. Bonneau, J., and Mironov, I. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems—CHES 2006* (2006), vol. 4249 of *Springer LNCS*, Springer, pp. 201–215.
10. Brumley, D., and Boneh, D. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 1–14.
11. Daemen, J., and Rijmen, V. *The Design of Rijndael*. Springer-Verlag, 2002.
12. Eidus, I., and Dickins, H. How to use the kernel samepage merging feature. https://www.kernel.org/doc/Documentation/vm/ksm.txt, November 2009.
13. 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.
14. Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Fine grain Cross-VM Attacks on Xen and VMware are possible. https://eprint.iacr.org/2014/248.pdf.
15. 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.
16. 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.
17. Neve, M. Cache-based vulnerabilities and spam analysis. *Doctor thesis, UCL* (2006).
18. of Standards, N. I., and Technology. Advanced encryption standard. *NIST FIPS PUB 197* (2001).
19. 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.
20. 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.
21. Suzaki, K., Iijima, K., Toshiki, Y., and Artho, C. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences 96*, 1 (2013), 215–224.
22. 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.
23. Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Software side channel attack on memory deduplication. *SOSP POSTER* (2011).
24. Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Effects of memory randomization, sanitization and page cache on memory deduplication.

25. The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, April 2003.
26. VMWare. Understanding memory resource management in vmware vsphere 5.0. http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf.
27. Waldspurger, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.
28. Wang, Z., and Lee, R. B. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 473–482.
29. 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.
30. Yarom, Y., and Benger, N. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. http://eprint.iacr.org/.
31. Yarom, Y., and Falkner, K. E. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive 2013* (2013), 448.
32. 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.