Cache Attacks Enable Bulk Key Recovery on the Cloud

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

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

Abstract. Cloud services keep gaining popularity despite the security concerns. While non-sensitive data is easily trusted to cloud, security critical data and applications are not. The main concern with the cloud is the shared resources like the CPU, memory and even the network adapter that provide subtle side-channels to malicious parties. We argue that these side-channels indeed leak fine grained, sensitive information and enable key recovery attacks on the cloud. Even further, as a quick scan in one of the Amazon EC2 regions shows, high percentage -55%- of users run outdated, leakage prone libraries leaving them vulnerable to mass surveillance.
The most commonly exploited leakage in the shared resource systems stem from the cache and the memory. High resolution and the stability of these channels allow the attacker to extract fine grained information. In this work, we employ the `Prime and Probe` attack to retrieve an RSA secret key from a co-located instance. To speed up the attack, we reverse engineer the cache slice selection algorithm for the Intel Xeon E5-2670 v2 that is used in our cloud instances. Finally we employ noise reduction to deduce the RSA private key from the monitored traces. By processing the noisy data we obtain the complete 2048-bit RSA key used during the decryption.

Keywords: Amazon EC2, Co-location Detection, RSA key recovery, Virtualization, Prime and Probe Attack.

1 Motivation

Cloud computing services are more popular than ever with their ease of access, low cost and real-time scalability. With increasing adoption of cloud, concerns over cloud specific attacks have been rising and so has the number of research studies exploring potential security risks in the cloud domain. A main enabler for cloud security is the seminal work of Ristenpart et al. [39]. The work demonstrated the possibility of co-location as well as the security risks that come with it. The co-location is the result of resource sharing between tenant Virtual Machines (VMs). Under certain conditions, the same mechanism can also be exploited to extract sensitive information from a co-located victim VM, resulting in

security and privacy breaches. Methods to extract information from VMs have been intensely studied in the last few years however remain infeasible within public cloud environments, e.g. [47, 36, 28, 40]. The potential impact of attacks on crypto processes can be even more severe, since cryptography is at the core of any security solution. Consequently, extracting cryptographic keys across VM boundaries has also received considerable attention lately. Initial studies explored the `Prime and Probe` technique on L1 cache [48, 19]. Though requiring the attacker and the victim to run on the same physical CPU core simultaneously, the small number of cache sets and the simple addressing scheme made the L1 cache a popular target. Follow up works have step by step removed restrictions and increased the viability of the attacks. The shared Last Level Cache (LLC) now enables true cross-core attacks [44, 8, 29] where the attacker and the victim share the CPU, but not necessarily the CPU core. Most recent LLC `Prime and Probe` attacks no longer rely on de-duplication [14, 25] or core sharing, making them more widely applicable.

With the increasing sophistication of attacks, participants of the cloud industry ranging from Cloud Service Providers (CSPs), to hypervisor vendors, up all the way to providers of crypto libraries have fixed many of the newly exploitable security holes through patches [1, 4, 3]—many in response to published attacks. However, many of the outdated cryptographic libraries are still in use, opening the door for exploits. A scan over the entire range of IPs in the South America East region yields that 55% of TLS hosts installed on Amazon EC2 servers have not been updated since 2015 and are vulnerable to an array of more recently discovered attacks. Consequently, a potential attacker such as a nation state, hacker group or a government organization can exploit these vulnerabilities for bulk recovery of private keys. Besides the usual standard attacks that target individuals, this enables mass surveillance on a population thereby stripping the network from any level of privacy. Note that the attack is enabled by our trust in the cloud. The cloud infrastructure already stores the bulk of our sensitive data. Specifically, when an attacker instantiates multiple instances in a targeted availability zone of a cloud, she co-locates with many vulnerable servers. In particular, an attacker trying to recover RSA keys can monitor the LLC in each of these instances until the pattern expected by the exploited hardware level leakage is observed. Then the attacker can easily scan the cloud network to build a public key database and deduce who the recovered private key belongs to. In a similar approach, Heninger et al. [21] scan the network for public keys with shared or similar RSA modulus factors due to poor randomization. Similarly Bernstein et al. [10] compiled a public key database and scanned for shared factors in RSA modulus commonly caused by broken random number generators.

In this work, we explore the viability of full RSA key recovery in the Amazon EC2 cloud. More precisely, we utilize the LLC as a covert channel both to co-locate and perform a cross-core side-channel attack against a recent cryptographic implementation. Our results demonstrate that even with complex and resilient infrastructures, and with properly configured random number generators, cache attacks are a big threat in commercial clouds.

Our Contribution

This work presents a full key recovery attack on a modern implementation of RSA in a commercial cloud and explores all steps necessary to successfully recover both the key and the identity of the victim. This attack can be applied under two different scenarios:

1. **Targeted Co-location:** In this scenario, we launch instances until we co-locate with the victim as described in [41, 24]. Upon co-location the secret is recovered by a cache enabled cross-VM attack.
2. **Bulk Key Recovery:** We randomly create instances and using cross-VM cache attacks recover imperfect private keys. These keys are subsequently checked and against public keys in public key database. The second step allows us to eliminate noise in the private keys and determine the identity of the owner of the recovered key.

Unlike in earlier bulk key recovery attacks [21, 10] we do not rely on faulty random number generators but instead exploit hardware level leakages.

Our specific technical contributions are as follows:

- We first demonstrate that the LLC contention based co-location detection tools are plausible in public clouds
- Second, we reverse-engineer the undocumented non-linear slice selection algorithm implemented in Intel Xeon E5-2670 v2 [2] used by our Amazon EC2 instances, and utilize it to automate and accelerate the attack
- Third, we describe how to apply the `Prime and Probe` attack to the LLC and obtain RSA leakage information from co-located VMs
- Last, we present a detailed analysis of the necessary post-processing steps to cope with the noise observed in a real public cloud setup, along with a detailed analysis on the CPU time (at most 30 core-hours) to recover both the noise-free key and the owner's identity (IP).

2 Related Work

This work combines techniques needed for co-location in a public cloud with state-of-the art techniques in cache based cross-VM side channel attacks.

Co-location detection: In 2009 Ristenpart et al. [39] demonstrated that a potential attacker has the ability to co-locate and detect co-location in public IaaS clouds. In 2011, Zhang et al. [46] demonstrated that a tenant can detect co-location in the same core by monitoring the L2 cache. Shortly after, Bates et al. [7] implemented a co-location test based on network traffic analysis. In 2015, Zhang et al. [47] demonstrated that de-duplication enables co-location detection from co-located VMs in PaaS clouds. In follow-up to Ristenpart et al.'s work [39], Zhang et al. [43] and Varadarajan et al. [41] explored co-location detection in commercial public cloud in 2015. Both studies use the memory bus contention

channel explored by Wu et al. in 2012 [42] to detect co-location. Finally in 2016, İnci et al. [24] explored co-location detection on Amazon EC2, Microsoft Azure and Google Compute Engine using three detection methods namely memory bus locking, LLC covert channel and LLC software profiling.

Recovering cache slice selection methods: A basic technique based on hamming distances for recovering and exploiting **linear** cache slice selection was introduced in [23]. Irazoqui et al. [26] and Maurice et al. [35] used a more systematic approach to recover linear slice selection algorithms in a range of processors, the latter pointing out the coincidence of the functions across processors. Recently, Yarom et al. [45] recovered a 6 core slice selection algorithm with a similar technique as the one presented in this work.

Side-channel attacks: RSA have been widely studied and explored with regards to diverse covert channels such as time [32, 12], power [31], EM emanations [15, 16], and even acoustic channels [17]. Micro-architectural side-channel attacks also succeeded on recovering secret keys used in symmetric cryptography. After the first theoretical [22, 38] and practical [9, 37] attacks utilizing micro-architectural covert channels, Aciicmez et al. [5, 6] demonstrated the possibility of recovering RSA secret keys using the instruction cache and the branch prediction unit as covert channels. Later, Zhang et al. [48] recovered El-Gamal secret keys across co-located VMs exploiting leakage in the upper level caches.

While all the previously mentioned side-channel attacks used a private core resource (i.e., attacker needs to be running in the same CPU core as the victim), in 2014 Yarom et al. [44] proved to be able to recover RSA secret keys across co-located VMs by using the `Flush and Reload` attack in the presence of memory de-duplication. Recently Liu et al. [14] implemented an attack against El-Gamal using the `Prime and Probe` attack in the LLC, whereas Bhattacharya et al. [11] utilized the branch prediction performance counters to recover RSA keys.

In addition to the attacks on public key cryptography schemes cache attacks also have been applied to AES [25, 29], ECDSA [8], TLS messages [30], the number of items in a shopping cart [47] or even the key strokes typed in a keyboard [18]. Even further, they recently have been applied to PaaS clouds [47], across processors [27] and in smartphones [34].

3 `Prime and Probe` in the LLC

In computer systems, the physical memory is protected and not visible to the user, who only sees the virtual addresses that the data resides at. Therefore a memory address translation stage is required to map virtual addresses to physical. However, there are some bits of the virtual address that remain untranslated, i.e, the least significant p_{low} bits with $2^{p_{low}}$ size memory pages. These are called the *page offset*, while the remaining part of the address is called the *page frame number* and their combination make the physical address. The location of a memory block in the cache is determined by its physical address. Usually the physical address is divided in three different sections to access *n-way* caches: the byte field, the set field and the tag field. The length of the byte and set fields

are determined by the cache line size and the number of sets in the cache, respectively. The more sets a cache has, more bits are needed from the *page frame number* to select the set that a memory block occupies in the cache.

The `Prime and Probe` attack has been widely studied in upper level caches [48, 5], but was first introduced for the LLC in [14, 25] with the use of **hugepages**. Unlike regular memory pages that reveal only 12 bits of the physical address, hugepages reveal 21 bits, allowing the LLC monitoring. Also, profiling the LLC in contrast to the L1 or L2 cache has various advantages. Firstly, unlike the upper level caches, the LLC is shared across cores, providing a cross-core covert channel. Moreover, the time distinguishability of accesses in upper level caches is much lower than those between the LLC and memory. On the other hand, due to the size of LLCs, we cannot simultaneously profile the whole cache, but rather a small portion of it at a time. In addition to that, modern processors divide their LLC into slices with a non-public hash algorithm, making it difficult to predict where the data will be located. Taking all these into account, the `Prime and Probe` attack is divided in two main stages:

Prime stage: The attacker fills a portion of the LLC with his own data and waits for a period of time for the victim to access the cache.

Probe stage: The attacker probes (reloads) the primed data. If the victim accessed the monitored set of the cache, one (or more) of the attacker's lines will not reside in the cache anymore, and will have to be retrieved from the memory.

As stated before, profiling a portion of the cache becomes more difficult when the LLC is divided into slices. However, as observed by [14] we can create an *eviction set* without knowing the algorithm implemented. This involves a step prior to the attack where the attacker finds the memory blocks colliding in a specific set/slice. This can be done by creating a large pool of memory blocks, and access them until we observe that one of them is fetched from the memory. The procedure will be further explained in section 4. A group of memory blocks that fill one set/slice in the LLC will form an *eviction set* for that set/slice.

4 Co-locating on Amazon EC2

In order to perform our experiments across co-located VMs we first need to make sure that they are running in the same server. We present the LLC as an exploitable covert channel with the purpose of detecting co-location between two instances. For the experiments, we launched 4 accounts (named A, B, C and D) on the South American Amazon EC2 region and launched 20 m3.medium instances in each of these accounts, 80 instances in total.

On these instances, we performed our LLC co-location detection test and obtained co-located instance pairs. In total, out of 80 instances launched from different accounts, we were able to obtain 7 co-located pairs and one triplet. Account A had 5 co-located instances out of 20 while B and C had 4 and 7 respectively. As for the account D, we had no co-location with instances from other accounts. Overall, assuming that the account A is the target, next 60 instances launched in accounts B, C, D have 8.3% probability of co-location

with the target. Note that all co-locations were between machines from different accounts. The experiments did not aim at obtaining co-location with a single instance, for which the probability of obtaining co-location would be lower.

4.1 The LLC Co-location Method

The LLC is shared across all cores of most modern Intel CPUs, including the Intel Xeon E5-2670 v2 used (among others) in Amazon EC2. Accesses to LLC are thus transparent to all VMs co-located on the same machine, making it the perfect domain for covert communication and co-location detection.

Our LLC test is designed to detect cache lines that are needed to fill a specific set in the cache. In order to control the location that our data will occupy in the cache, the test allocates and works with hugepages.[1] In normal operation with moderate noise, the number of lines to fill one set is equal to LLC associativity, which is 20 in Intel Xeon E5-2670 v2 used in our Amazon EC2 instances. However, with more than one user trying to fill the same set at the same time, one will observe that fewer than 20 lines are needed to fill one set. By running this test concurrently on a co-located VM pair, both controlled by the same user, it is possible to verify co-location with high certainty. The test performs the following steps:

- Prime one memory block b_0 in a set in the LLC
- Access additional memory blocks $b_1, b_2, \ldots, b_n$ that occupy the same set, but can reside in a different slice.
- Reload the memory block b_0 to check whether it has been evicted from the LLC. A high reload time indicates that the memory block b_0 resides in the RAM. Therefore we know that the required m memory blocks that fill a slice are part of the accessed additional memory blocks $b_1, b_2, \ldots, b_n$.
- Subtract one of the accessed additional memory blocks b_i and repeat the above protocol. If b_0 is still loaded from the memory, b_i does not reside in the same slice. If b_0 is now located in the cache, it can be concluded that b_i resides in the same cache slice as b_0 and therefore fill the set.
- Count the number of memory blocks needed to fill a set/slice pair. If the number is significantly different than the associativity, it can be concluded that we have cache contention across co-located VMs.

The LLC is not the only method that we have tried in order to verify co-location (see Appendix for more information). However, the experiments show that the LLC test is the only decisive and reliable test that can detect whether two of our instances are running in the same CPU in Amazon EC2. We performed the LLC test in two steps as follows:

[1] The co-location test has to be implemented carefully, since the heavy usage of hugepages may yield into performance degradation. In fact, while trying to achieve a quadruple co-location Amazon EC2 stopped our VMs due to performance issues. For a more detailed explanation see Appendix.

1. **Single Instance Elimination:** The first step of the LLC test is the elimination of single instances i.e. the ones that are not co-located with any other in the instance pool. To do so, we schedule the LLC test to run at all instances at the same time. Instances not detecting co-location is retired. For the remaining ones, the pairs need to be further processed as explained in the next step. Note that without this preliminary step, one would have to perform $n(n-1)/2$ pair detection tests to find co-located pairs, i.e. 3160 tests for 80 instances. This step yielded 22 co-located instances out of 80.
2. **Pair Detection:** Next we identify pairs for the possibly co-located instances. The test is performed as a binary search tree where each instance is tested against all the others for co-location.

4.2 Challenges and Tricks of Co-location Detection

During our experiments on Amazon EC2, we have observed various problems and interesting events related to the underlying hardware and software. Here we discuss what to expect when experimenting on Amazon EC2.

Instance Clock Decay: In our experiments using Amazon EC2, we have noticed that instance clocks degrade slowly over time. More interestingly, after detecting co-location using the LLC test, we discovered that co-located instances have the same clock degradation with 50 nanoseconds resolution. We believe that this information can be used for co-location detection.

Hardware Complexity: Modern Amazon EC2 instances have much more advanced and complex hardware components like 10 core, 20 thread CPUs and SSDs. Thus, our cache profiling techniques have to be adapted to handle servers with multiple slices that feature non-linear slice selection algorithms.

Co-located VM Noise: Compute cloud services including Amazon EC2 maintain a variety of services and servers. Most user-based services, however, quiet down when users quiet down, i.e. after midnight. Especially between 2 a.m. and 4 a.m. Internet traffic as well as computer usage is significantly lower than the rest of the day. We confirmed this assumption by measuring LLC noise in our instances and collected data from 6 instances over the course of 4 week days. Results are shown in Figure 1. LLC noise and thus server load are at its peak around 8 p.m. and lowest at 4 a.m. We also measured the noise observed in the first 200 sets of the LLC for one day in Figure 2. The y-axis shows the probability of observing a cache access by a co-located user other than victim during a `Prime and Probe` interval of the spy process (i.e. the attacker cannot detect the cache access of the victim process). The measurements were taken every 15 minutes. A constant noise floor at approx. 4.5% is present in all sets. Sets 0 and 3 feature the highest noise, but high noise (11%)is observed at the starting points of other pages as well. In fact, certain set numbers (0,3,26,39,58) mod 64 seem to be predictably more noisy and not well suited for the attack.

Dual Socket machines: We did not find evidence of dual socket machines among the medium instances that we used in both co-location and attack steps. Indeed once co-located, our LLC co-location test always succeeded over time,

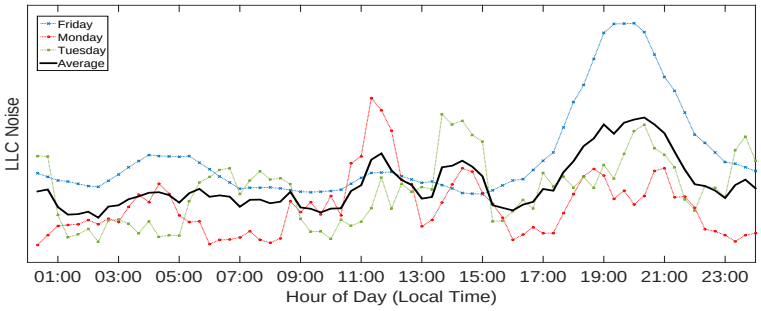

Fig. 1. LLC Noise over time of day, by day (dotted lines) and on average (bold line).

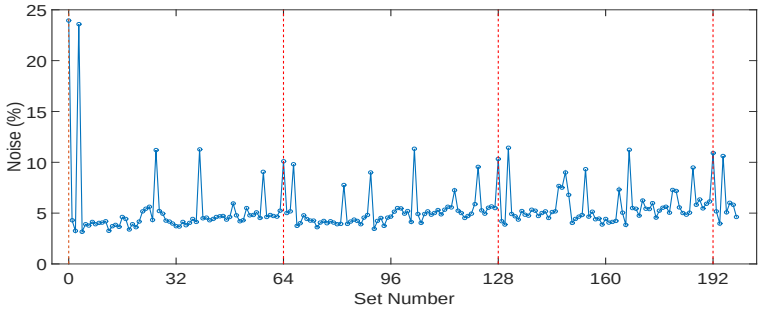

Fig. 2. Average noise for the first 200 sets in a day. Red lines are the starting points of pages. Sets 0 and 3 feature the highest amount of noise, with a repeating pattern every 64 sets (which is the width of a page in the LLC).

even after a year. If the instances were to reside in dual socket machines and the VM processes moved between CPUs, the co-location test would have failed. However, even in that case, repeated experiments would still reveal co-location just by repeating the test after a time period enough to allow a socket migration.

5 Obtaining The Non-linear Slice Selection Algorithm

The LLC attack that will later be performed is based on the ability of generating colliding memory blocks, i.e., blocks that collide for a specific set and slice. In modern processors, each set in the LLC is divided into slices (usually one slice per core) to respond to multiple requests at a time. The usage of a sliced LLC as a covert channel becomes simpler when we deal with a power of two number of slices. In these cases, due to the linearity, the set bits does not affect the slice bits in the eviction set created for one of the slices. Thus, we could create an eviction set for a specific set-slice pair composed by $b1, b2, ..bn$ memory blocks choosing a random set s. If we later want to target a different set, we could still use $b1, b2, ...bn$ by changing only the set bits and they will fill the same slice. This fact was used in [14, 25] to perform LLC side channel attacks. This peculiarity is

not observed in non-linear slices, i.e., the same $b1, b2, .., bn$ will only slice-collide for a small number of sets. The slice colliding blocks can either be empirically observed for each set, or guessed if the non-linear slice selection algorithm is known. Our particular EC2 instance type utilizes a Intel Xeon E5-2670 v2, which features a 25MB LLC distributed over 10 LLC slices (i.e., non power of two). We decide to reverse-engineer the non-linear slice selection algorithm to speed up our eviction set creation algorithm. Note that the approach that we follow can be utilized to reverse engineer *any* non-linear slice selection algorithm.

We describe the slice selection algorithm as

$$H(p) = h_3(p) \| h_2(p) \| h_1(p) \| h_0(p) \tag{1}$$

where each $H(p)$ is a concatenation of 4 different functions corresponding to the 4 necessary bits to represent 10 slices. Note that $H(p)$ will output results from 0000 to 1001 if we label the slices from 0-9. Thus, a non-linear function is needed that excludes outputs 10-15. Further note that p is the physical address and will be represented as a bit string: $p = p_0 p_1 \dots p_{35}$. In order to recover the non-linear hash function implemented by the Intel Xeon E5-2670 v2, we use a fully controlled machine featuring the same Intel Xeon E5-2670 v2 found in Amazon's EC2 servers. We first generate ten equation systems (one per slice) based on slice colliding addresses by applying the same methodology explained to achieve co-location and generating up to 100,000 additional memory blocks.

Up to this point, one can solve the non-linear function after a re-linearization step given sufficiently many equations. Since we are not be able to recover enough addresses (due to RAM limitations) we take a different approach. Figure 3 shows the distribution of the 100,000 addresses over the 10 slices. Note that 8 slices are mapped to by 81.25% of the addresses, while 2 slices get only about 18.75%, i.e., a 3/16 proportion. We will refer to these two slices as the *non-linear slices*.

We proceed to first solve the first 8 slices and the last 2 slices separately using linear functions. For each we try to find solutions to the equation systems

$$P_i \cdot \hat{H}_i = \hat{0}, \tag{2}$$

$$P_i \cdot \hat{H}_i = \hat{1}\ . \tag{3}$$

Here P_i is the equation system obtained by arranging the slice colliding addresses into a matrix form, $\hat{H}_i$ is the matrix containing the slice selection functions and $\hat{0}$ and $\hat{1}$ are the all zero and all one solutions, respectively. This outputs two sets of linear solutions both for the first 8 linear slices and the last 2 slices.

Given that we can model the slice selection functions separately using linear functions, and given that the distribution is non-uniform, we model the hash function is implemented in two levels. In the first level a non-linear function chooses between either of the 3 linear functions describing the 8 linear slices or the linear functions describing the 2 non-linear slices. Therefore, we speculate that the 4 bits selecting the slice looks like:

$$H(p) = \begin{cases} h_0(p) = h_0(p) & h_1(p) = \neg(nl(p)) \cdot h_1'(p) \\ h_2(p) = \neg(nl(p)) \cdot h_2'(p) & h_3(p) = nl(p) \end{cases}$$

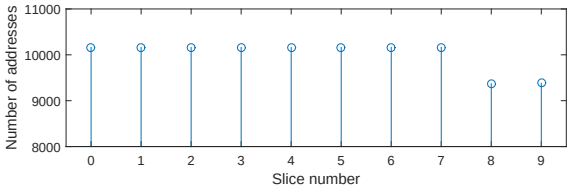

Fig. 3. Number of addresses that each slice takes out of 100,000. The non-linear slices take less addresses than the linear ones.

Table 1. Results for the hash selection algorithm implemented by the Intel Xeon E5-2670 v2

f	**Hash function** $H(p) = h_0(p)\|\neg(nl(p)) \cdot h'_1(p)\|\neg(nl(p)) \cdot h'_2(p)\|nl(p)$
h_0	$p_{18} \oplus p_{19} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{30} \oplus p_{32} \oplus p_{33} \oplus p_{34}$
h'_1	$p_{18} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
h'_2	$p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{26} \oplus p_{28} \oplus p_{30}$
nl	$v_0 \cdot v_1 \cdot \neg(v_2 \cdot v_3)$
v_0	$p_9 \oplus p_{14} \oplus p_{15} \oplus p_{19} \oplus p_{21} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{29} \oplus p_{32} \oplus p_{34}$
v_1	$p_7 \oplus p_{12} \oplus p_{13} \oplus p_{17} \oplus p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{25} \oplus p_{27} \oplus p_{31} \oplus p_{32} \oplus p_{33}$
v_2	$p_9 \oplus p_{11} \oplus p_{14} \oplus p_{15} \oplus p_{16} \oplus p_{17} \oplus p_{19} \oplus p_{23} \oplus p_{24} \oplus p_{25} \oplus p_{28} \oplus p_{31} \oplus p_{33} \oplus p_{34}$
v_3	$p_7 \oplus p_{10} \oplus p_{12} \oplus p_{13} \oplus p_{15} \oplus p_{16} \oplus p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{30}$ $\oplus p_{31} \oplus p_{32} \oplus p_{33} \oplus p_{34}$

where h_0,h_1 and h_2 are the hash functions selecting bits 0,1 and 2 respectively, h_3 is the function selecting the 3rd bit and nl is a nonlinear function of an unknown degree. We recall that the proportion of the occurrence of the last two slices is 3/16. To obtain this proportion we need a degree 4 nonlinear function where two inputs are negated, i.e.:

$$nl = v_0 \cdot v_1 \cdot \neg(v_2 \cdot v_3) \tag{4}$$

Where nl is 0 for the 8 linear slices and 1 for the 2 non-linear slices. Observe that nl will be 1 with probability 3/16 while it will be zero with probability 13/16, matching the distributions seen in our experiments. Consequently, to find v_0 and v_1 we only have to solve Equation (3) for slices 8 and 9 together to obtain a 1 output. To find v_2 and v_3, we first separate those addresses where v_0 and v_1 output 1 for the linear slices $0-7$. For those cases, we solve Equation (3) for slices $0-7$. The result is summarized in Table 1. We show both the non-linear function vectors v_0, v_1, v_2, v_3 and the linear functions h_0, h_1, h_2. These results describe the behavior of the slice selection algorithm implemented in the Intel Xeon E5-2670 v2. With this result, we can now easily predict the slice selection on the target processor in the EC2 cloud.

6 Cross-VM RSA Key Recovery

To prove the viability of the `Prime and Probe` attack in Amazon EC2 across co-located VMs, we present an expanded version of the attack implemented in [14] by showing its application to RSA. It is important to remark that the attack *is not* processor specific, and can be implemented in any processor with inclusive last level caches. In order to perform the attack:

- We make use of the fact that the offset of the address of each table position entry does not change when a new decryption process is executed. Therefore, we only need to monitor a subsection of all possible sets, yielding a lower number of traces.
- Instead of the monitoring both the multiplication and the table entry set (as in [14] for El-Gamal), we *only monitor a table entry set in one slice*. This avoids the step where the attacker has to locate the multiplication set and avoids an additional source of noise.

The attack targets a sliding window implementation of RSA-2048 where each position of the pre-computed table will be recovered. We will use Libgcrypt 1.6.2 as our target library, which not only uses a sliding window implementation but also uses CRT and message blinding techniques [33]. The message blinding process is performed as a side channel countermeasure for `chosen-ciphertext` attacks, in response to studies such as [17, 16].

We use the `Prime and Probe` side channel technique to recover the positions of the table T that holds the values $c^3, c^5, c^7, \ldots, c^{2^W-1}$ where W is the window size. For CRT-RSA with 2048 bit keys, $W = 5$ for both exponentiations d_p, d_q. Observe that, if all the positions are recovered correctly, reconstructing the key is a straightforward step.

Recall that we do not control the victim's user address space. This means that we do not know the location of each of the table entries, which indeed changes from execution to execution. Therefore we will monitor a set hoping that it will be accessed by the algorithm. However, our analysis shows a special behavior: each time a new decryption process is started, even if the location changes, the offset field does not change from decryption to decryption. Thus, we can *directly* relate a monitored set with a specific entry in the multiplication table.

The knowledge of the processor in which the attack is going to be carried out gives an estimation of the probability that the set/slice we monitor collides with the set/slice the victim is using. For each table entry, we fix a specific set/slice where not much noise is observed. In the Intel Xeon E5-2670 v2 processors, the LLC is divided in 2048 sets and 10 slices. Therefore, knowing the lowest 12 bits of the table locations, we will need to monitor *one* set/slice that solves $s \bmod 64 = o$, where s is the set number and o is the offset for a table location. This increases the probability of probing the correct set from $1/(2048 \cdot 10) = 1/20480$ to $1/((2048 \cdot 10)/64) = 1/320$, reducing the number of traces to recover the key by a factor of 64. Thus our spy process will monitor accesses to *one* of the 320 set/slices related to a table entry, hoping that the RSA encryption accesses it

when we run repeated decryptions. Thanks to the knowledge of the non linear slice selection algorithm, we can easily change our monitored set/slice if we see a high amount of noise in one particular set/slice. Since we also have to monitor a different set per table entry, it also helps us to change our eviction set accordingly. The threshold is different for each of the sets, since the time to access different slices usually varies. Thus, the threshold for each of the sets has to be calculated before the monitoring phase. In order to improve the applicability of the attack the LLC can be monitored to detect whether there are RSA decryptions or not in the co-located VMs as proposed in [24]. After it is proven that there are RSA decryptions the attack can be performed.

In order to obtain high quality timing leakage, we synchronize the spy process and the RSA decryption by initiating a communication between the victim and attacker, e.g. by sending a TLS request. Note that we are looking for a particular pattern observed for the RSA table entry multiplications, and therefore processes scheduled before the RSA decryption will not be counted as valid traces. In short, the attacker will communicate with the victim before the decryption. After this initial communication, the victim will start the decryption while the attacker starts monitoring the cache usage. In this way, we monitor 4,000 RSA decryptions with the same key and same ciphertext for each of the 16 different sets related to the 16 table entries.

We investigate a hypothetical case where a system with dual CPU sockets is used. In such a system, depending on the hypervisor CPU management, two scenarios can play out; processes moving between sockets and processes assigned to specific CPUs. In the former scenario, we can observe the necessary number of decryption samples simply by waiting over a longer period of time. In this scenario, the attacker would collect traces and only use the information obtained during the times the attacker and the victim share sockets and discard the rest as missed traces. In the latter scenario, once the attacker achieves co-location, as we have in Amazon EC2, the attacker will always run on the same CPU as the target hence the attack will succeed in a shorter span of time.

7 Leakage Analysis Method

Once the online phase of the attack has been performed, we proceed to analyze the leakage observed. There are three main steps to process the obtained data. The first step is to identify the traces that contain information about the key. Then we need to synchronize and correct the misalignment observed in the chosen traces. The last step is to eliminate the noise and combine different graphs to recover the usage of the multiplication entries. Among the 4,000 observations for each monitored set, only a small portion contains information about the multiplication operations with the corresponding table entry. These are recognized because their exponentiation trace pattern differs from that of unrelated sets. In order to identify where each exponentiation occurs, we inspected 100 traces and created the timeline shown in Figure 4(b). It can be observed that the first exponentiation starts after 37% of the overall decryption time. Note that among

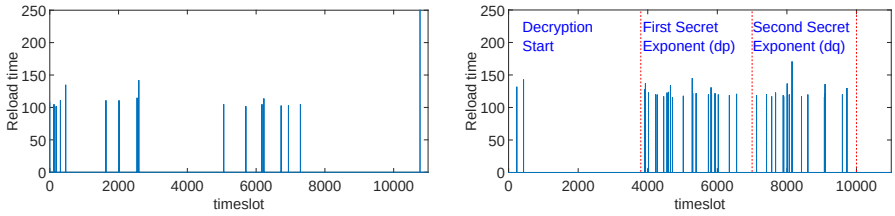

Fig. 4. Different sets of data where we find a) trace that does not contain information b) trace that contains information about the key

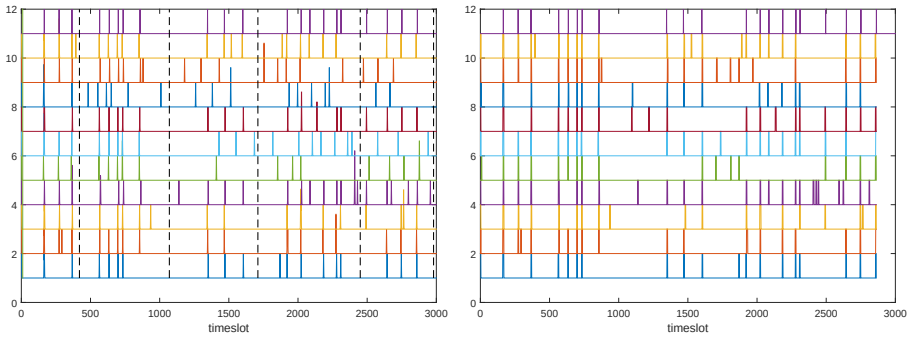

Fig. 5. 10 traces from the same set where a) they are divided into blocks for a correlation alignment process b) they have been aligned and the peaks can be extracted

all the traces recovered, only those that have more than 20 and less than 100 peaks are considered. The remaining ones are discarded as noise. Figure 4 shows measurements where no correct pattern was detected (Fig. 4(a)), and where a correct pattern was measured (Fig. 4(b)).

In general, after the elimination step, there are $8-12$ correct traces left per set. We observe that data obtained from each of these sets corresponds to 2 consecutive table positions. This is a direct result of CPU cache prefetching. When a cache line that holds a table position is loaded into the cache, the neighboring table position is also loaded due to cache locality principle.

For each graph to be processed, we first need to align the creation of the look-up table with the traces. Identifying the table creation step is trivial since each table position is used twice, taking two or more time slots. Figure 5(a) shows the table access position indexes aligned with the table creation. In the figure, the top graph shows the true table accesses while the rest of the graphs show the measured data. It can be observed that the measured traces suffer from misalignment due to noise from various sources e.g. RSA or co-located neighbors.

To fix the misalignment, we take most common peaks as reference and apply a correlation step. To increase the efficiency, the graphs are divided into blocks and processed separately as seen in Figure 5(a). At the same time, Gaussian

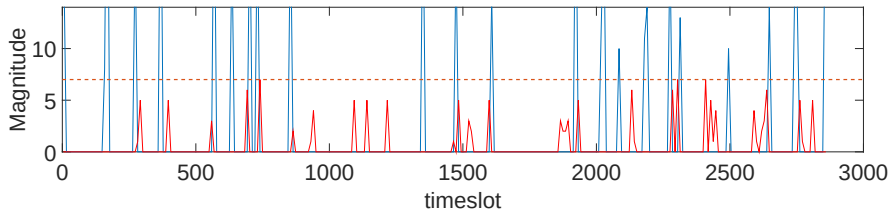

Fig. 6. Eliminating false detections using a threshold (red dashed line) on the combined detection graph.

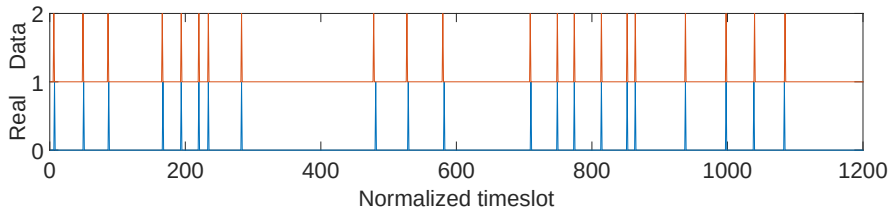

Fig. 7. Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution

filtering is applied to peaks. In our filter, the variance of the distribution is 1 and the mean is aligned to the peak position. Then for each block, the cross-correlation is calculated with respect to the most common hit graph i.e. the intersection set of all graphs. After that, all graphs are shifted to the position where they have the highest correlation and aligned with each other. After the cross-correlation calculation and the alignment, the common patterns are observable as in Figure 5(b). Observe that the alignment step successfully aligns measured graphs with the true access graph at the top, leaving only the combining and the noise removal steps. We combine the graphs by simple averaging and obtain a single combined graph.

In order to get rid of the noise in the combined graph, we applied a threshold filter as can be seen in Figure 6. We used 35% of the maximum peak value observed in graphs as the threshold value. Note that a simple threshold was sufficient to remove noise terms since they are not common between graphs.

Now we convert scaled time slots of the filtered graph to real time slot indexes. We do so by dividing them with the spy process resolution ratio, obtaining the Figure 7. In the figure, the top and the bottom graphs represent the true access indexes and the measured graph, respectively. Also, note that even if additional noise peaks are observed in the obtained graph, it is very unlikely that two graphs monitoring consecutive table positions have noise peaks at the same time slot. Therefore, we can filter out the noise stemming from the prefetching

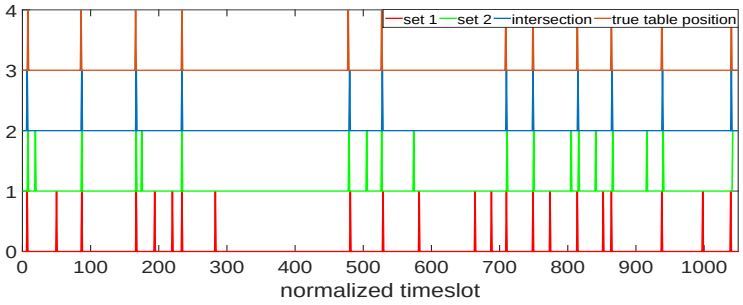

Fig. 8. Combination of two sets

Table 2. Successfully recovered peaks on average in an exponentiation

Average Number of traces/set	4000
Average number of correct graphs/set	10
Wrong detected peaks	7.19%
Missdetected peaks	0.65%
Correctly detected peaks	92.15%

by combining two graphs that belong to consecutive table positions. Thus, the resulting indexes are the corresponding timing slots for look-up table positions.

The very last step of the leakage analysis is finding the intersections of two graphs that monitor consecutive sets. By doing so, we obtain accesses to a single table position as seen in Figure 8 with high accuracy. At the same time, we have total of three positions in two graphs. Therefore, we also get the positions of the neighbors. A summary of the result of the leakage analysis is presented in Table 2. We observe that more than 92% of the recovered peaks are in the correct position. However, note that by combining two different sets, the wrong peaks will disappear with high probability, since the chance of having wrong peaks in the same time slot in two different sets is very low.

8 Recovering RSA Keys with Noise

We divide the section in two different scenarios, i.e., the scenario where the identity and public key of the target is known (targeted co-location) and the scenario where we have no information about the public key (bulk key recovery).

8.1 Targeted Co-location: The Public Key is Known

In this case we assume that the attacker implemented a targeted co-location against a known server, and that she has enough information about the public key parameters of the target. The leakage analysis described in the previous

section recovers information on the CRT version of the secret exponent d, namely $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$. A noise-free version of either one can be used to trivially recover the factorization of $N = pq$, since $\gcd(m - m^{ed_p}, N) = p$ for virtually any m [13].

In cases where the noise on d_p and d_q is too high for a direct recovery with the above-mentioned method, their relation to the known public key can be exploited if the used public exponent e is small [20].

Almost all RSA implementations currently use $e = 2^{16} + 1$ due to the heavy performance boost over a random and full size e. For CRT exponents it holds that $ed_p = 1 \mod (p-1)$ and hence $ed_p = k_p(p-1) + 1$ for some $1 \leq k_p < e$ and similarly for d_q, yielding $k_p p = ed_p + k_p - 1$ and $k_q p = ed_q + k_q - 1$.

Algorithm 1 Windowed RSA Key Recovery with Noise

```
for k_p from 1 to e − 1 do
    Compute k_q = (1 − k_p)(k_p N − k_p + 1)^{−1} (mod e)
    while i < |wp| do
        Process windows wp[i], wp[i + 1]
        Introduce shifts; vary ip[i]] up max_zeros
        for each d_p variation do
            Compute X = Σ_{j=0}^{i+1} wp[j]2^{ip[j]}
            Identify wq that overlap with wp[i], wp[i + 1]
            Compute Y = Σ_{j=0}^{i+1} wq[j]2^{iq[j]}
            if δ(X, Y, t)=0 then
                Update wp, ip, wq, iq
                Create thread for i + 1
            end if
            if if no check succeeded then
                too many failures: abandon thread.
                if max_zeros achieved then
                    i = i − 1
                end if
                Update ip, wq, iq
                Create thread for i
            end if
        end for
    end while
end for
```

Multiplying both equations gives us a key equation which we will exploit in two ways

$$k_p k_q N = (ed_p + k_p - 1)(ed_q + k_q - 1). \tag{5}$$

If we consider Equation (5) modulo e, the unknowns d_p and d_q disappear and we obtain $k_p k_q N = (k_p - 1)(k_q - 1) \pmod{e}$. Therefore given k_p we can recover k_q and vice versa by solving this linear equation. Since $1 \leq k_p < e$ represents

an exhaustible small space we can simply try all values for k_p and compute corresponding k_q as shown above.

Next, assume we are given the first t bits of d_p and d_q, e.g. $a = d_p \mod 2^t$ and $b = d_q \mod 2^t$. For each k_p we check whether $\delta(a, b, t) = 0$ where

$$\delta(a, b, t) = k_p k_q N - (ea + k_p - 1)(eb + k_q - 1) \pmod{2^t}$$

This means we have a simple technique to check the correctness of the least-significant t bits of d_p, d_q for a choice of k_p. We can

- **Check parts** of d_p and d_q by verifying if the test $\delta(d_p(t), d_q(t), t) = 0$ holds for $t \in [1, \lceil \log(p) \rceil]$.
- **Fix alignment and minor errors** by shifting and varying $d_p(t)$ and $d_q(t)$, and then sieving working cases by checking if $\delta(d_p(t), d_q(t), t) = 0$,
- **Recover parts** of d_q given d_p (and vice versa) by solving the error equation $\delta(d_p(t), d_q(t), t) = 0$ in case the data is missing or too noisy to correct.

Note that the algorithm may need to try all 2^{16} values of k_p in a loop. Further, in the last case where we recover a missing data part using the checking equation we need to speculatively continue the iteration for a few more steps. If we observe too many mistakes we may early terminate the execution thread without reaching the end of d_p and d_q.

To see how this approach can be adapted into our setting, we need to consider the error distribution observed in d_p and d_q as recovered by cache timing. Furthermore, since the sliding window algorithm was used in the RSA exponentiation operation, we are dealing with variable size (1-5 bit) *windows* with contents wp, wq, and window positions ip, iq for d_p and d_q, respectively.

The windows are separated by 0 strings. We observed:

- The window wp contents for d_p had no errors and were in the correct order. There were slight misalignments in the window positions ip with extra or missing zeros in between.
- In contrast, d_q had not only alignment problems but also few windows with incorrect content, extra windows, and missing windows (overwritten by zeros). The missing windows were detectable since we do not expect unusually long zero strings in a random d_q.
- Since the iterations proceed from the most significant windows to the least we observed more errors towards the least significant words, especially in d_q.

Algorithm 1 shows how one can progressively error correct d_p and d_q by processing groups of consecutive ℓ windows of d_p. The algorithm creates new execution threads when an assumption is made, and kills a thread after assumptions when too many checks fail to produce any matching on different windows. However, then the kill threshold has to be increased and the depth of the computation threads and more importantly the number of variations that need to be tested increases significantly. In this case, the algorithm finds the correct private key in the order of microseconds for a noise-free d_p and needs 4 seconds for our recovered d_p.

8.2 Bulk Key Recovery: The Public Key is Unknown

In this scenario, the attacker spins multiple instances and monitors the LLC, looking in all of them for RSA leakages. If viable leakages are observed, the attacker might not know the corresponding public key. However, she can build up a database of public keys by mapping the entire IP range of the targeted Amazon EC2 region and retrieve all the public keys of hosts that have the TLS port open. The attacker then runs the above described algorithm for each of the recovered private keys and the entire public key database. Having the list of 'neighboring' IPs with an open TLS port also allows the attacker to initiate TLS handshakes to make the servers use their private keys with high frequency.

In the South America Amazon EC2 region, we have found 36000+ IP addresses with the TLS port open (Appendix Figure **??**) using `nmap`. With a public key database of that size, our algorithm takes between less than a second (for noise-free d_ps) and 30 CPU hours (noisy d_ps) to check each private key with the public key database. This approach recovers the public/private key pair, and consequently, the identity of the key owner.

9 Countermeasures

Libgcrypt 1.6.3 update: Libgcrypt recently patched this vulnerability by making the sliding window multiplication table accesses indistinguishable from each other. Thus, an update to the latest version of the library avoids the leakage exploited in this work albeit only for ciphers using sliding window exponentiation.

Single-tenant Instances: Although more expensive, in most cloud services, users have the option of having the whole physical machine to themselves, preventing co-location with potentially malicious users.

Live Migration: In a highly noisy environment like the commercial cloud, an attacker would need many traces to conduct a side-channel attack. In the live migration scenario, the attacker would have to perform the attack in the time period when the attacker and the victim share the physical machine.

10 Conclusion

In conclusion, we show that even with advanced isolation techniques, resource sharing still poses security risk to public cloud customers that do not follow the best security practices. The cross-VM leakage is present in public clouds and can be a practical attack vector for data theft. Therefore, users have a responsibility to use latest improved software for their critical cryptographic operations. Even further, we believe that smarter cache management policies are needed both at the hardware and software levels to prevent side-channel leakages.

11 Acknowledgments

This work is supported by the National Science Foundation, under grants CNS-1318919 and CNS-1314770.

References

1. Fix Flush and Reload in RSA. https://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000329.html.
2. Intel Xeon 2670-v2. http://ark.intel.com/es/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz.
3. OpenSSL fix flush and reload ECDSA nonces. https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=2198be3483259de374f91e57d247d0fc667aef29.
4. Transparent Page Sharing: additional management capabilities and new default settings. http://blogs.vmware.com/security/vmware-security-response-center/page/2.
5. Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.
6. Aciiçmez, O., K. Koç, c., and Seifert, J.-P. Predicting secret keys via branch prediction. In *Topics in Cryptology CT-RSA 2007*, vol. 4377. pp. 225–242.
7. 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*.
8. 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.
9. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.
10. Bernstein, D. J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., and Van Someren, N. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology-ASIACRYPT 2013*. Springer, 2013, pp. 341–360.
11. Bhattacharya, S., and Mukhopadhyay, D. Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. In *CHES 2015*.
12. Brumley, D., and Boneh, D. Remote Timing Attacks are Practical. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 1–14.
13. Campagna, M. J., and Sethi, A. Key recovery method for CRT implementation of rsa. Cryptology ePrint Archive, Report 2004/147. http://eprint.iacr.org/.
14. Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level cache side channel attacks are practical. In *S&P 2015*.
15. Gandolfi, K., Mourtel, C., and Olivier, F. Electromagnetic analysis: Concrete results. In *CHES 2001*, vol. 2162 of *Lecture Notes in Computer Science*. pp. 251–261.
16. Genkin, D., Pachmanov, L., Pipman, I., and Tromer, E. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *CHES* (2015), Lecture Notes in Computer Science, Springer, pp. 207–228.
17. Genkin, D., Shamir, A., and Tromer, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pp. 444–461.
18. Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.
19. Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. SP '11, pp. 490–505.

20. Hamburg, M. Bit level Error Correction Algorithm for RSA Keys. Personal Communication, Cryptography Research, Inc., 2013.
21. Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 205–220.
22. Hu, W.-M. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*.
23. Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pp. 191–205.
24. İncI, M. S., Gülmezoglu, B., Eisenbarth, T., and Sunar, B. Co-location detection on the cloud. In *COSADE* (2016).
25. Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *36th IEEE Symposium on Security and Privacy (S&P 2015)*.
26. Irazoqui, G., Eisenbarth, T., and Sunar, B. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Euromicro DSD* (2015).
27. Irazoqui, G., Eisenbarth, T., and Sunar, B. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security* (2016), ASIA CCS '16, ACM.
28. Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies 1(1)*, 25–40.
29. Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.
30. Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS '15, pp. 85–96.
31. Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in Cryptology CRYPTO 99*, vol. 1666 of *Lecture Notes in Computer Science*. pp. 388–397.
32. Kocher, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113.
33. Libgcrypt. The Libgcrypt reference manual. `http://www.gnupg.org/documentation/manuals/gcrypt/`.
34. Lipp, M., Gruss, D., Spreitzer, R., and Mangard, S. Armageddon: Last-level cache attacks on mobile devices. *CoRR abs/1511.04897* (2015).
35. Maurice, C., Scouarnec, N. L., Neumann, C., Heen, O., and Francillon, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters . In *RAID 2015* (2015).
36. Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1406–1418.
37. 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*, CT-RSA'06.
38. Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.

39. 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*, CCS '09, pp. 199–212.
40. 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* (2013), 215–224.
41. Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 913–928.
42. Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium* (2012), pp. 159–173.
43. Xu, Z., Wang, H., and Wu, Z. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 929–944.
44. 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)*, pp. 719–732.
45. Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. Mapping the intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. `http://eprint.iacr.org/`.
46. 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*.
47. Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.
48. 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*.