Seriously, get off my cloud!
Cross-VM RSA Key Recovery in a Public Cloud

Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, Berk Sunar
Worcester Polytechnic Institute, Worcester, MA, USA
Email:{msinci, bgulmezoglu, girazoki, teisenbarth, sunar}@wpi.edu

***Abstract*—It has been six years since Ristenpart et al. [29] demonstrated the viability of co-location and provided the first concrete evidence for sensitive information leakage on a commercial cloud. We show that co-location can be achieved and detected by monitoring the last level cache in public clouds. More significantly, we present a full-fledged attack that exploits subtle leakages to recover RSA decryption keys from a co-located instance. We target a recently patched Libgcrypt RSA implementation by mounting Cross-VM *Prime and Probe* cache attacks in combination with other tests to detect co-location in Amazon EC2. In a preparatory step, we reverse engineer the unpublished nonlinear slice selection function for the 10 core Intel Xeon processor which significantly accelerates our attack (this chipset is used in Amazon EC2). After co-location is detected and verified, we perform the Prime and Probe attack to recover noisy keys from a carefully monitored Amazon EC2 VM running the aforementioned vulnerable libgcrypt library. We subsequently process the noisy data and obtain the complete 2048-bit RSA key used during encryption. This work reaffirms the privacy concerns and underlines the need for deploying stronger isolation techniques in public clouds.**

Amazon EC2, Co-location Detection, RSA key recovery, Resource Sharing, Prime and Probe

I. 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 services, 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-specific security analysis is the seminal work of Ristenpart et al. [29]. The work demonstrated the possibility of co-location as well as the security risks that come with co-location. The co-location detection can be enabled and detected by 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 victim VMs have been intensely studied in the last few years however infeasible within public cloud environments, e.g. see [37], [28], [21], [30]. 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 [38], [16]. 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 eliminated restrictions and increased the viability of the attacks. Last level cache (LLC) attacks now enable true cross-core attacks [34], [8], [22] where the attacker and the victim share the same CPU, but not necessarily the same core. Most recent LLC prime and probe attacks no longer rely on deduplication [11], [19], making them more widely applicable and harder to prevent.

With the increasing sophistication of attacks, participants of the cloud industry ranging from Cloud Service Providers, 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. Specifically, RSA, El-Gamal and AES cipher implementations of OpenSSL, Libgcrypt, PolarSSL and CyaSSL were updated to prevent cache attacks. However, most of the recent works have focused on improving accuracy and applicability of attack techniques in controlled lab environments.

In this work, we explore the current state of security in public IaaS clouds and possibility of a cryptographic key recovery attack by mounting a cache attack across two instances (both controlled by the attacker) co-located in Amazon EC2. While the attack is still possible, our results show that, through combined efforts of all involved parties, the bar for performing successful attacks in the cloud is quickly rising. Specifically, we show that many co-location techniques that gave high accuracy in 2009 no longer work. Similarly, increased hardware complexity and better protected cryptographic libraries increase the cost and required sophistication of attacks to succeed. Nevertheless, many covert channels are still exploitable and require further patches. We highlight these remaining issues by what we believe is the first work in the literature to succeed in a full-blown key recovery attack on a recent cryptographic implementation in a commercial IaaS cloud.

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 perform such an attack. In particular, the work first revisits the co-location problem. Our experiments show that the co-location techniques presented in [29] have been addressed by AWS and no longer are a significant indicator for co-location. In addition, we present new techniques that show that co-location is still detectable in 2015 by using other

shared resources such as LLCs. Once co-located, we exploit the LLC to recover the secret key of a modern sliding-window exponentiation based implementation of RSA, across cores and without relying on deduplication. We expand beyond the techniques presented in [11] and [19] and show that a detailed study of the LLC structure of the targeted processor yields a much more efficient attack. In particular, we reverse engineer the LLC non-linear slice selection algorithm of Intel Xeon E5-2670 v2 chipset, which is dominant on Amazon EC2. Finally, we present several techniques necessary for a cache attack to succeed in a public cloud. Most notably, we present methods to filter noise from the observed leakage at two levels: (i) by using an alignment filter to reduce temporal noise introduced by co-residing processes and the OS, (ii) by rejecting false detections using a threshold technique on the aligned traces. Finally we present an error correction algorithm that exploits dependencies between the public key and the observed private key to remove remaining errors and to recover error free RSA keys.

A combination of the new co-location technique with the advanced attack methods presented in this work highlights the practicality and potential risk of cache attacks. The results urge providers of cryptographic libraries to update their code to ensure that cache based leakages no longer are exploitable, as recently done by Libgcrypt in their 1.6.3 release. In summary, this work describes how to retrieve information from co-located VMs in public clouds. We divide the description into 4 main sections:

- We first demonstrate that previous presented techniques to detect co-location do not work anymore and present novel techniques to detect co-location across cores in a public cloud
- Second, we obtain knowledge of the undocumented non-linear slice selection algorithm implemented in Intel Xeon E5-2670 v2 [2], the most common processor used in Amazon EC2. This knowledge is beneficial to adapt our spy process to accelerate the attack
- Third, we describe how to apply `Prime and Probe` to obtain RSA leakage information from co-resident VMs
- Last, we present a detailed analysis of the necessary post processing steps to recover the noise free RSA key.

II. Prime and Probe in the LLC

In order to understand the `Prime and Probe` attack, we first need to review the cache placement techniques in modern processors. First of all, physical memory is protected and not visible to the user, who only sees the virtual addresses that his data obtains. Therefore a memory translation stage is needed to map virtual to physical addresses. 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 portion of bits are called the *page offset*, the remaining translated bits are 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. Note that the bigger the amount of sets, the more bits that 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 [38], [5], but was first introduced for the LLC in [11], [19] thanks to the usage of huge size pages. Indeed, one of the main reasons why `Prime and Probe` was never applied in last level caches is that regular pages limit the amount of knowledge of the physical address, being only able to infer the data location in small caches. With huge size pages, the user has knowledge of the lowest 21 bits of the physical address, being able to profile and monitor bigger caches such as the LLC.

Profiling the LLC has many advantages over the upper level caches. For instance, upper level caches are not shared across cores, thereby limiting the practicality of the attack to core co-resident tenants. However, last level caches are shared across cores and a suitable covert channel for cross-core attacks. Moreover, the access time difference between upper level caches is much lower than the difference between LLC and memory. Therefore, LLC side channel attacks have more resolution and are less affected by noise.

On the other hand, last level caches are much bigger. Therefore we can not profile the whole cache, but we have to focus in a small portion of it. In addition to that, modern processors divide their LLC into slices with a non-public hash algorithm, making more difficult to predict where the data will be located.

The `Prime and Probe` attack is divided in two main stages:

Prime stage: In this step, the attacker fills a portion of the LLC with his own created data and waits for a specified period of time to detect if someone accesses the cache.

Probe stage: In this step, the attacker probes (reloads) the primed data. If someone accessed the monitored of the cache, one (or more) of his lines will not reside in the cache anymore, and will have to be retrieved from the memory. This is translated in a bigger probe time than when the attacker retrieves all his data from the cache, i.e, when no one accessed the cache during his monitorization activity.

As stated before, profiling a portion of the cache becomes more difficult when the LLC is divided into slices. However, as observed by [11] 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 (observing a higher reload time). Then we simply identify the memory blocks that created that eviction and group them. We will refer to the group of memory blocks that fill one set/slice in the LLC as the *eviction set* for that set/slice.

III. Co-locating on Amazon EC2

The first step prior to implementing any cross-core side channel attack in a public cloud is to show the ability to co-locate two different instances in the same hardware. For that purpose, we first revisit the methods used in [29] and present new techniques to detect co-location in up-to-date Amazon EC2. For the experiments, we launched 4 accounts (named A, B, C and D) on Amazon EC2 and launched 20 m3.medium instances in each of these accounts, 80 instances in total. We will refer to these instances as A12, B1, C20, D15 etc., numbered according to their public IP addresses.

Using these four accounts, we performed our LLC co-location detection test and obtained co-located instance pairs. We even achieved a triple co-location meaning that 3 instances from 3 different accounts co-located on the same physical machine. However, as explained in Section III-C, these three instances were scheduled for retirement shortly after the co-location tests. In total, out of 80 instances launched from 4 different accounts around same time, 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 account D, we had no co-located instances. Overall, our experiments show that, when we assume the account A is the target, next 60 instances launched in accounts B, C, D have 8.3% probability of co-location. 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.

In the following, we explore different methods to detect co-location. Using the LLC test as our ground truth for co-location, we revisited known methods used in [29] and evaluated the viability of new methods. We also believe that even though the methods explained below do not work on Amazon EC2 (other than the LLC test), they can be applied to other public clouds where stronger leakages still exist.

A. Revisiting Known Detection Methods

In [29], the authors use Hypervisor IP, Instance IP, Ping Test and Disk Drive Test as tools for co-location detection. We have tried these methods as well and found that, Amazon in fact did a good job of fixing these attack vectors and they are no longer useful.

Hypervisor IP: Using the `traceroute` tool, we collected first hop IP addresses from our instances. The idea behind this collection is that instances located in same physical machine should have the same first hop address, presumably the hypervisor IP. However, experiments show that there are only few different first hop IP addresses used for large number of instances, only four for our 80 instances. Also, with repeated measurements we noticed that these addresses were actually dynamic, rather than assigned IPs. Even further, we later confirmed by LLC test results that co-located instances do not share the same first hop address, making this detection method useless.

Instance IP: Like [29], we also checked for any possible algebraic relation or proximity between our instance IP addresses. After detecting co-located instances with the LLC test, we checked both internal and external IP addresses of co-located instances and concluded that IP address assignment is random and does not leak any information about co-location in Amazon EC2 anymore.

Ping Test: In a network, ping delay between two nodes depend on various factors such as network adapters of nodes, network traffic and most importantly the number of hops between the nodes. In [29], authors used this information to detect co-location on the assumption that co-located instances have shorter ping delays. By sending pings from each instance to all other 80, we obtained round trip network delays for all instances. From each account we sent 20 repeated pings and obtained maximum, average and minimum ping times. We discarded the maximum ping values since they are highly effected from network traffic and do not provide reliable information. Average and minimum ping values on the other hand are more directly related to the number of hops between two instances. While co-location correlates with lower ping times, it fails to provide conclusive evidence for co-location. Figure 3 shows the heat map of our ping timings, dark blue indicating lower and red representing high round trip times. Also, x and y axes represent ping source and target instances respectively. Visual inspection of the figure reveals: (i) Diagonal representing the self-ping (through external IP) time is clearly distinguishable and low compared to the rest of the targets; (ii) Network delay of the source instance affects the round trip time significantly, requiring an in depth analysis to find relatively close instances; (iii) Last 20 instances that belong to account D have significantly lower and uniform overall ping delays than the rest of the accounts.

In order to eliminate the delay stemming from the source instance, we decided to find the 3 closest neighbors of each instance rather than applying a constant threshold. After filtering strong outliers, we used multiple datasets with average and minimum ping times to create a more reliable metric. For different datasets, consistent closest neighbors to an instance indicate either co-location or sharing the same subnet. Using three datasets taken at different times of day, we created Figures 1 and 2 that show consistent close neighbors according to average and minimum ping times respectively, for all 80 instances. As seen from figures, network is highly susceptible to noise which consistency of ping times significantly. Apart from the instance 37 in Figure 1, no instance has consistent low ping neighbors nearly enough to suspect a co-location. In conclusion, even though the ping test reveals some information about proximity of instance networks, as seen from self ping times, it is not fine grain enough to be used for co-location detection.

Disk Drive Benchmark: To replicate [29]'s disk drive test, we used the `dd` tool due to its file size and repetition flexibility. `dd` is a Unix command line utility used to copy files, backup disks and perform disk benchmarks. In our performance degradation

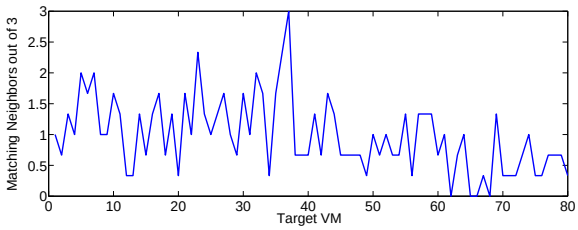

Fig. 1. Consistent Neighbors According to the Average Ping Time

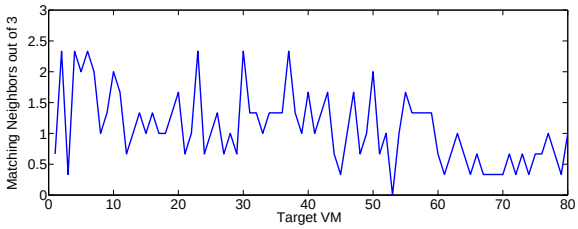

Fig. 2. Consistent Neighbors According to the Minimum Ping Time

tests, we used `dd` as a tool to measure disk performance under simultaneous heavy load from multiple instances and use the results to detect possible co-locations. Using `dd`, we repeatedly wrote various size blocks of zeros to the disk drive and measured the write speed. Also, in order to maximize the performance degradation, we tried various file sizes ranging from 1 KB to 2 GB to find an optimal detection point. Instances from all four accounts were scheduled to perform the test at the same time. In order to achieve synchrony, we updated instance times to `time.nist.gov` prior to each test. The aim of this test was to observe any potential performance degradations in disk write speeds due to two or more co-located instances.

Our `dd` test results are presented in Figure 4 where the horizontal and vertical axis represent the instance number and the disk write speed in MBps, respectively. Note that, we did not include any results for files smaller than 1 MB since `dd` does not provide stable results for such files. Figure 4 clearly shows that disk performance is almost constant for all instances, including the co-located ones. This means that either all 80 instances are using separate disk drives or more likely that every instance has its disk access bandwidth limited to a certain value to prevent bottlenecks. In conclusion, the performance degradation in co-located instances is unstable, negligibly low at about $2 \sim 3$ percent, and thus inconclusive.

B. New Detection Methods

Since the previously explored detection methods show very low significance in our experiments, we explored additional covert channels that can potentially be used to detect instance co-location.

LLC Test: 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 the number of 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 huge size pages [19], [11]. 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 Amazon EC2. However, with more than one user trying to fill the same set at the same time, one will notice 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.

As a result of this continuous cache line creation, when multiple tests are executed concurrently on a system (and therefore priming the same set), the test detects the high number of accesses to the L3 cache by the line creation stage of the other test and outputs high amount of cache lines that evict the given cache set. Keep in mind that other processes running in the system can create false positives as well, but it does not affect the test output in a significant way and the established covert channel can still detect co-location with high accuracy.

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) **Single Instance Elimination:** The first step of the LLC test is the elimination of single instances i.e. instances that are not co-located with any instance in the instance pool. In order to do so, we scheduled the LLC test to run at all instances at the same time. Any instance detecting no 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 our 80 instances. This step yielded 22 possibly co-located instances out of

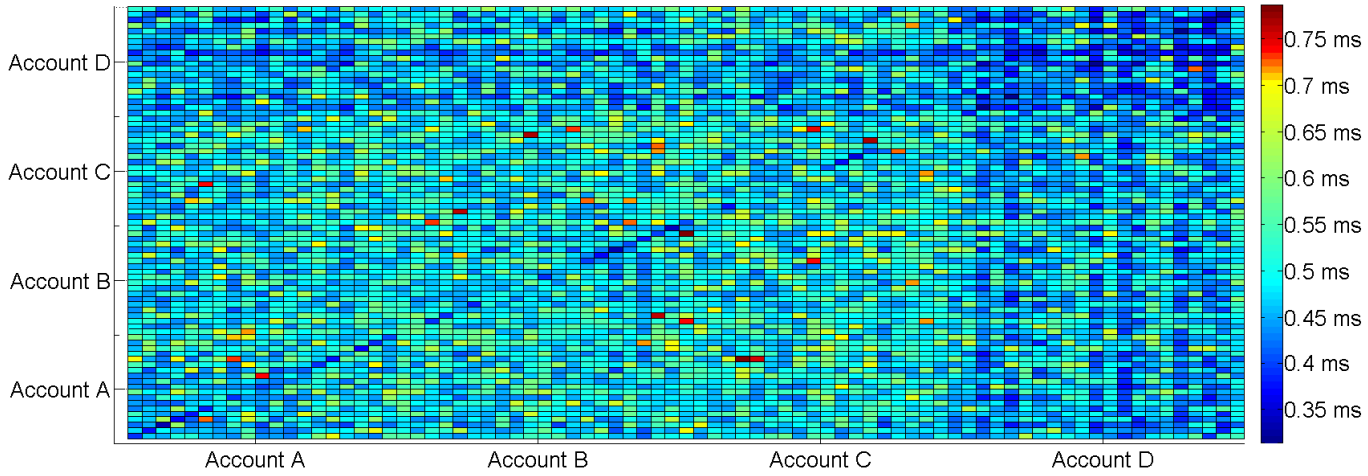

Fig. 3. Ping time heat map for all 80 instances created using minimum ping times for each source instance

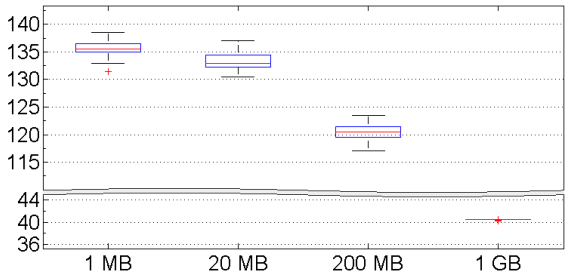

Fig. 4. `dd` performance test results for various data sizes; 2 GB file copied 2 times, 200 MB file copied 5 times, 20 MB file copied 50 times, 1 MB file copied 1000 times

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.

CPU Benchmark: To create a bottleneck at the CPU level, we used `Hardinfo` CPU benchmark suite. The suite provides a wide range of benchmarks, namely CryptoHash, Fibonacci number calculation, N-Queens test, FFT calculation and Ray-tracing. However, our experiments show that the instance isolation and the resources management in Amazon EC2 prevent this test from creating any performance degradation and therefore does not provide the required stable, high resolution results.

AES-NI Benchmark: AES-NI is the high performance AES hardware module found in most modern Intel processors including the ones used in Amazon EC2. The Intel Xeon E5-2670 v2 datasheet [2] does not specify whether each core has its own module or all cores use a single AES-NI module. We suspected that by creating bottlenecks in the shared AES-NI module we could detect co-location. However, our experiments revealed that the AES-NI modules are not shared between cores and each CPU core uses its own module, making this method useless for cross-core detection.

C. Challenges and Tricks of Co-location Detection

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

Instance Clock Decay: In our experiments at 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 0.05 milliseconds 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 compared to dual core CPUs and magnetic HDDs used in [29]. Even further, these SSDs are managed by the hypervisor specifically to allow multiple read/write operations to run concurrently without creating bottlenecks, making performance degradation much harder to observe.

Hypervisor Hardware Obfuscation: In reaction to [29], Amazon has fixed information leakages about the underlying hardware by modifying their Xen Hypervisor. Currently, no sensor data such as fan speed, CPU and system temperature or hardware MAC address is revealed to instances. Serial numbers and all other hardware identifiers are either emulated or censored, mitigating any co-location detection using this information.

Co-located VM Noise: Amazon EC2 is a versatile ecosystem where customers perform wide range of computations and provide services. Noise created by these co-located instances

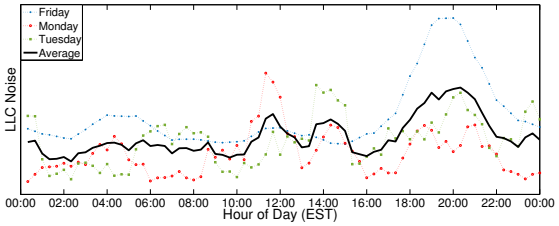

Fig. 5. LLC Noise over time

is hard to filter but not unpredictable. In order to cope with this noise, two tricks can be used; running experiments at low noise hours and using low noise instance types.

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 5. As expected, LLC noise and thus server load are at its peak around 8 p.m. and lowest at 4 a.m. Low usage hours present valuable low noise time window for both achieving co-location and running the co-location checks. During these low noise hours, very few additional instances are launched. Fewer new instances translate to higher chance of co-locating with a specific target.

Secondly, instance type selection also affects noise in experiments. In Amazon EC2, each instance has an IP address and network connectivity whether it has 1 core or 16 cores. So when a physical machine hosts 16 m3.medium single core instances, it also supports 16 network connections. Reversely, a physical machine hosting 2 m3.2xlarge instances with 8 cores each, it only supports network connections for the two. Making the reasonable assumption that 16 instances create more network traffic than 2, we believe that the ping test is affected by the instance types hosted on the physical machine. The probability of co-locating with an 8 core instance with less noise is much lower than with a 2 core instance. In an experimental co-location scenario, the trade-off between these options must be carefully considered.

Clock Differences Between Instances: In order to run performance degradation tests simultaneously, OS clocks in instances must be synchronized. In order to assure synchronization during tests, we have updated the system times using `ntpdatetime.nist.gov` command multiple times during experiments to keep the synchronization intact. Note that without time synchronization, we have observed differences of system times between instances of up to a minute.

Dual Socket machines: We did not find any evidence of dual socket machine existence for the medium instances that we used in both co-location and attack steps. Indeed once co-located, our co-location test always succeeded over time, even after three months. If our instances were to be placed in dual socket machines, the co-location test would fail when the attacker and the victim VMs were running in different sockets. 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. Even further, after the co-location is detected, the RSA key recovery attack that will later be explained would succeed as well only by increasing the number of traces.

Instance Retirement: A very interesting feature of Amazon EC2 is instance retirement in case of perceived hardware failure or overuse. Through constant resource monitoring EC2 detects significant performance degradation on one of the hardware components such as disk drive, network adapter or a GPU card in a physical system, and marks the instance for retirement. If there is no malfunction or hazardous physical damage to the underlying hardware, e-mail notifications are sent to all users who have instances running on the physical machine. If there is such an immediate hardware problem, instances are retired abruptly and a notification e-mail is sent to users afterwards. We observed this behavior on our triple co-located instances (across three accounts). While running our performance tests to create a bottleneck and determine co-location, we received three separate e-mails from Amazon to the three involved accounts notifying us that our instances A5, B7 and C7 had a hardware failure and are scheduled for instance retirement. The important thing to note here is that, via our tests, we have previously determined that these instances A5, B7 and C7 are co-located on the same physical machine. We assume that our performance based co-location tests were the cause of the detected performance degradation in the system that raised flags with Amazon EC2 health monitoring system, resulting in the instance retirement.

Placement Policy: In our experiments, instances launched within short time intervals of each other were more likely to be co-located. To exploit the placement policy and increase chances of co-location, one should launch multiple instances in a close time interval with the target. Note that two instances from the same account are never placed on the same physical machine. While this increases the probability of co-locating with a victim in the in the actual attack scenario, it also makes it harder to achieve co-location for experiments.

IV. Obtaining The Non-linear Slice Selection Algorithm

Once co-location was achieved in Amazon EC2, we want to check the possibility of implementing a cross-core LLC side channel attack between co-located VMs. Unlike in [11], [19] where the attacks run in systems which implement linear slice selection algorithms (i.e power of two number of slices), Amazon EC2 medium instances use Intel Xeon E5-2670 v2 machines, which implement a non-linear LLC slice selection algorithm. It is important to note that `Prime and Probe` attacks become much simpler when linear slice selection algorithms are used, because the memory blocks to create an eviction set for different set values do not change. This means that we can calculate the eviction set for, e.g, set 0 and the

memory blocks will be the same if we profile a different set s. As we will see in this section, this is not true for non-linear slice selection algorithms (where the profiled set also affects the slice selected). This fact makes the table and multiplication function location finding step implemented in [11], [19] much more difficult and time consuming. Although knowing the slice selection algorithm implemented is not crucial to run a `Prime and Probe` attack (since we could calculate the eviction set for every set s that we want to monitor), the knowledge of the non-linear slice selection algorithm can save significant time, specially when we have to profile a big number of sets. Indeed, in the attack step, we will select a range of sets/slices $s_1, s_2, ...s_n$ for which, thanks to the knowledge of the non-linear slice selection algorithm, we know that the memory blocks in the eviction set will not change.

Hund et al. [18] utilized a method based on a comparison of hamming distances between varying addresses to reverse engineer a 4 slice Sandy Bridge processor. The reverse engineered Sandy Bridge slice selection algorithm turned out to use only the address bits not involved in the set and byte fields. Indeed, this seems to be true for all processors which have a *linear* slice selection method, i.e., the number of slices is a power of two. Liu et al. [11] and Irazoqui et al. [19] used this fact to perform LLC side channel attacks.

However, this is not true when the number of slices is not a power of two. The Intel Xeon E5-2670 v2, the most widely used EC2 instance type, has a 25MB LLC distributed over 10 LLC slices. By just performing some small tests we can clearly observe that the set field affects the slice selection algorithm implemented by the processor. Indeed, it is also clearly observable that the implemented hash function is a *non-linear* function of the address bits, since the 16 memory blocks mapped to the same set in a huge memory page cannot be evenly distributed over 10 slices. Thus 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 perform experiments in a fully controlled machine featuring the same Intel Xeon E5-2670 v2 found in Amazon's EC2 servers. We first generate ten equation systems based on addresses colliding in the same slice by applying the same methodology explained to achieve co-location and generating up to 100,000 additional memory blocks. We repeat the same process 10 times, changing the primed memory block b_0 in each of them to target a different slice. This outputs 10 different systems of addresses, each one referring to a different slice.

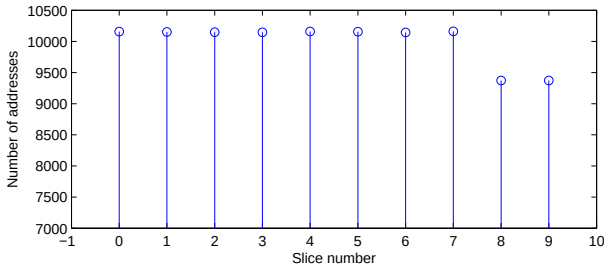

Fig. 6. Number of addresses that each slice takes out of 100,000. The non-linear slices take less addresses than the linear ones.

The first important observation we made on the 10 different systems is that 8 of them behave differently from the remaining 2. In 8 of the address systems recovered, if 2 memory blocks *in the same huge memory page* collide in the same slice, they only differ in the 17th bit. This is not true for the remaining two address systems. We suspect, at this point, that the 2 systems behaving differently are the 8th and 9th slice. We will refer to these two slices as the non-linear slices.

Up to this point, one can solve the non-linear function after a re-linearization step given sufficiently many equations. However, one may not be able to recover enough addresses. Recall that the higher the degree of the non-linear term the more equations are needed. In order to keep our analysis simpler we decided to take a different approach. The second important observation we made is on the distribution of the addresses over the 10 slices. It turns out that the last two slices are mapped to with a lower number of addresses than the remaining 8 slices. Figure 6 shows the distribution of the 100,000 addresses over the 10 slices. The different distributions seen for the last two slices give us evidence that a non-linear slice selection function is implemented in the processor. Even further, it can be observed that the linear slices are mapped to by 81.25% of the addresses, while the non-linear slices get only about 18.75%. The proportion is equal to $3/16$. We will make use of this uneven distribution later.

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 a linear solutions both for the first 8 linear slices and the last 2 slices separately.

Given that we can model the slice selection functions separately using linear functions, and given that the distribution is non-uniform, we suspect that 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 look like:

$$H(p) = \begin{cases} h_0(p) = h_0(p) \\ h_1(p) = \neg(nl(p)) \cdot h'_1(p) \\ h_2(p) = \neg(nl(p)) \cdot h'_2(p) \\ h_3(p) = nl(p) \end{cases}$$

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 distribution 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 I. 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. It can be observed that the bits involved in the set selection (bits 6 to 16 for the LLC) are also involved in the slice selection process, unlike with linear selection algorithms. This means that for different sets, different memory blocks will map to the same slice. However, with this result, we can now easily predict the slice selection on the target processor in the AWS cloud.

Note that the method applied here can be used to reverse engineer other machines that use different non-linear slice selection algorithms. By looking at the distribution of the memory blocks over all the slices, we can always get the shape of the non-linear part of the slice selection algorithm. The rest of the steps are generic, and can be even applied for linear slices selection algorithms.

V. Cross-VM RSA Key Recovery

To prove the viability of `Prime and Probe` attacks in Amazon EC2 across co-located VMs, we present an expanded version of the attack implemented in [11] 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 that allows the allocation of huge size pages. There are some main differences between this work and [11]:

- 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 when compared to [11].
- In contrast to the work in [11] in which both the multiplication and the table entry set are monitored, we *only monitor a table entry set in one slice*. This avoids the step where the attacker has to locate the multiplication set, thereby avoiding 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 [26]. The message blinding process is performed as a side channel countermeasure for `chosen-ciphertext` attacks, in response to studies such as [14], [13]. A description of the RSA decryption process implemented by Libgcrypt is shown in Algorithm 1. The sliding window implementation that Libgcrypt uses is explained in Algorithm 2, where a table holds values $c^3, c^5, c^7, \ldots, c^{2^W-1}$.

Algorithm 1 RSA with CRT and Message Blinding

Input: $c \in \mathbb{Z}_N$, Exponents d, e, Modulus $N = pq$
Output: m
$r \xleftarrow{\$} Z_N$ with $\gcd(r, N) = 1$ ▷ Message Blinding
$c^* = c \cdot r^e \mod N$
$d_p = d \mod (p-1)$ ▷ CRT conversion
$d_q = d \mod (q-1)$
$m_1 = (c^*)^{d_p} \mod p$ ▷ Modular Exponentiation
$m_2 = (c^*)^{d_q} \mod q$
$h = q^{-1} \cdot (m_1 - m_2) \mod p$ ▷ Undo CRT
$m^* = m_2 + h \cdot q$
$m = m^* \cdot r^{-1} \mod N$ ▷ Undo Blinding
return m

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.

Furthermore, the knowledge of the processor in which the attack is going to be carried out is important to implement the attack. 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 every set that solves $s \mod 64 = o$, where s is the set number and o is the offset for a table location. This increases the

TABLE I
RESULTS FOR THE HASH SELECTION ALGORITHM IMPLEMENTED BY THE INTEL XEON E5-2670 V2

Vector	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	$p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{26} \oplus p_{28} \oplus p_{30}$
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	$p_7 \oplus p_{10} \oplus p_{12} \oplus p_{13} \oplus p_{15} \oplus p_{16} \oplus p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{30} \oplus p_{31} \oplus p_{32} \oplus p_{33} \oplus p_{34}$
nl	$v_0 \cdot v_1 \cdot \neg(v_2 \cdot v_3)$

Algorithm 2 RSA Sliding-Window Exponentiation

Input: Ciphertext $c \in \mathbb{Z}_N$, Exponent d, Window Size w
Output: $c^d \mod N$
$T[0] = c^3 \mod N$ ▷ Table Precomputation
$v = c^2 \mod N$
for i **from** 1 **to** $2^{w-1} - 1$ **do**
  $T[i] = T[i-1] \cdot v \mod N$
end for
$b = 1, j = len(d)$ ▷ Exponentiation Step
while $j > 0$ **do**
  if $e_j == 0$ **then**
    $b = b^2 \mod N$
    $j = j - 1$
  else
    Find $e_j e_{j-1} \ldots e_l \mid j - l + 1 \leq w$ with $e_l = 1$
    $b = b^{2^{j-l+1}} \mod N$
    if $e_j == 1$ **then**
      $b = b \cdot c \mod N$
    else
      $b = b \cdot T[(e_i - 3)/2] \mod N$
    end if
    $j = j - l - 1$
  end if
end while
return b

Algorithm 3 Spy Process Algorithm

for i from 0 to $2^{w-1} - 1$ **do**
  Find Set_Offset($T[i]$)
end for
timeslots=0
while timeslots¡maximum **do**
  prime
  wait
  t=*probe*
  if $t_i$threshold **then**
    Access[timeslots]=true
  else
    Access[timeslots]=false
  end if
end while

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. Recall that the address of the table entry varies from one encryption to the other (although with the same offset). 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 encryptions. Thanks to the knowledge of the non linear slice selection algorithm, we will select a range of sets/slices $s_1, s_2, \ldots, s_n$ for which the memory blocks that create the eviction sets do not change, and that allow us to profile all the precomputed table entries. A short description of the spy process is shown in Algorithm 3. 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 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.

Finally, in a hypothetical case where a dual socket system is used and VM processes move between sockets, our attack would still succeed. By only increasing the number of traces collected, we can obtain the same leakage information. 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.

VI. 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

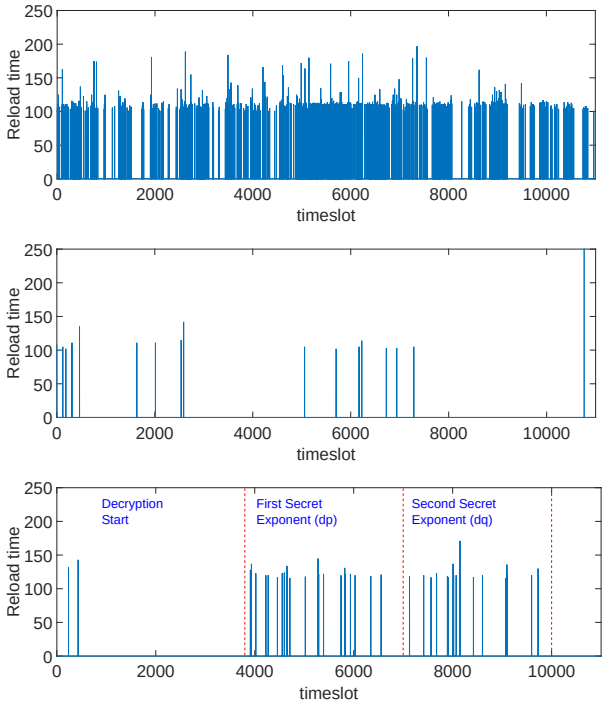

Fig. 7. Different sets of data where we find a) noisy trace b) trace that does not contain information c) trace that contains information about the key

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 will be recognized because their exponentiation trace pattern differs from that of unrelated sets. In order to identify where each exponentiation occurs in an RSA decryption, we inspected 100 traces and created the timeline shown in Figure 7(c). It can be observed that the first exponentiation starts after 37% of the overall encryption time. Note that, among 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 7 shows measurements where only noise was detected (Fig. 7(a)), no correct pattern was detected (Fig. 7(b)), and where a correct pattern was measured (Fig. 7(c)).

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 8(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, co-located neighbors etc.

For fixing 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 8(a). At the same time, Gaussian 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 reference 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 8(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 9. 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 10. 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 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 11 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 II. 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.

VII. Recovering RSA Keys with Noise

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

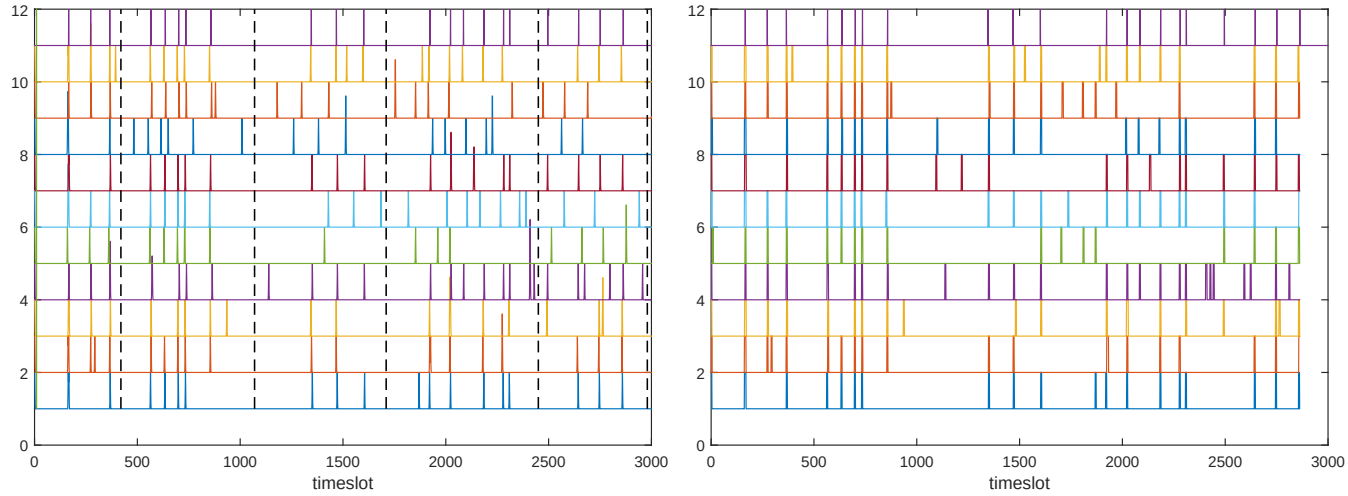

Fig. 8. 10 traces from the same set where a) they are divided into blocks to perform a correlation alignment process b) they have been aligned and the peaks should be extracted

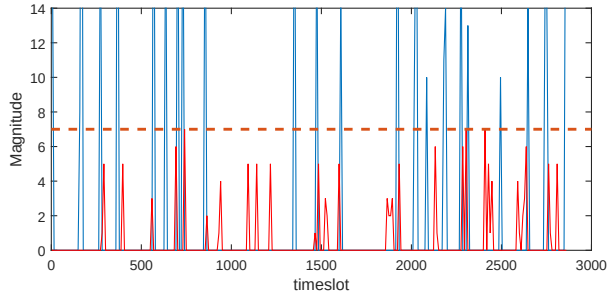

Fig. 9. Eliminating false detections using a threshold (red dashed line) on the combined detection graph.

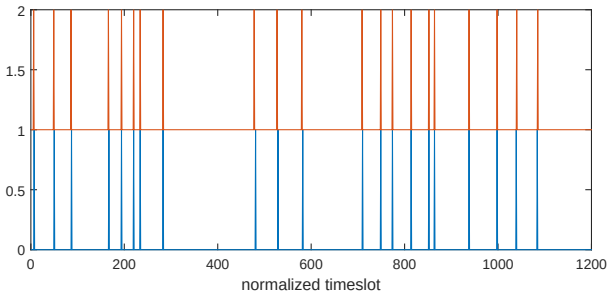

Fig. 10. Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution

TABLE II
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%

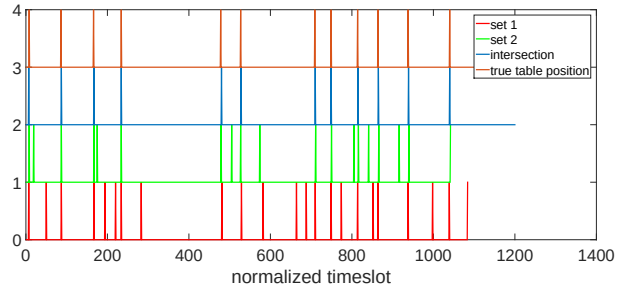

Fig. 11. Combination of two sets

the factorization of $N = pq$, since $\gcd(m - m^{ed_p}, N) = p$ for virtually any m [10]. However, 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 [17].

Almost all RSA implementations currently use $e = 2^{16} + 1$ due to the significant performance boost over a random and full size e. For the 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 $k_q p = ed_q + k_q - 1$. 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.

Next, assume we are given the first t-bits of d_p and d_q, e.g. $d_p(t)$ and $d_q(t)$. 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. For each k_p we check

whether $\delta(d_p(t), d_q(t), t) = 0$ 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 zeroes 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 zeroes). 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 4 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. In practice, it does suffice to use only a few windows, e.g $\ell = 2$ or 3. The higher ℓ the fewer assumptions will be made. 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.

Algorithm 4 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 + ℓ]
        Introduce shifts; vary ip[i], ..., ip[i + ℓ]
        for each d_p variation do
            Compute X = Σ_{j=0}^{i+ℓ} wp[j] 2^{ip[j]}
            Identify wq that overlap with wp[i], ..., wp[i+ℓ]
            Vary iq and wq for insertions/deletions
            Set t as end of wp[i + ℓ]
            for each d_q variation do
                Compute Y = Σ_{j=0}^{i+ℓ'} wq[j] 2^{iq[j]}
                if δ(X, Y, t)=0 then
                    Update wp, ip, wq, iq
                    Create thread for i + ℓ
                end if
            end for
            if if no check succeeded then
                too many failures abandon tread
                for each d_q variation do
                    solve Y from δ(X, Y, t)=0
                    Update wp, ip, wq, iq
                    Create thread for i + ℓ
                end for
            end if
        end for
    end while
end for
```

VIII. Countermeasures

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

Disabling Hugepages: Our `Prime and Probe` attack as well as the LLC detection method accelerates by exploiting huge sized pages. Therefore disabling hugepages on hypervisor level would significantly slow down malicious adversaries.

A More Sophisticated Instance Placement Policy: As explained in [39], co-location resistant placement algorithms can reduce the chances of co-location and protect customers from potential adversaries.

Single-tenant Instances: Placing multiple instances of a user on the same physical machine prevents co-location with a malicious attacker. Most cloud service providers including Amazon EC2 offer single tenant instances albeit as an expensive option. This option offers a number of benefits including isolation from other users.

Live Migration: In a highly noisy environment like the commercial cloud, an attacker would need many traces to conduct a side-channel attack. Live migration reaps the attacker from luxury of having no time constraints and makes targeting a specific instance harder. In the live migration scenario, the attacker would have to first detect co-location and then carry

out an actual attack before either one of the attacker or the target moves to another physical machine.

LLC Isolation: LLC is the most commonly used shared resource between VMs. Compared to disk drives and network adapters it is incredibly fast, allowing connection of large amounts of data to be transferred in a short time. Also, unlike the IP detection method, it does not require a pre-detection stage which makes it the perfect attack vector. This is why LLC isolation between CPU cores can mitigate both co-location detection and the cache side channel attacks in the cloud.

IX. 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. [29] demonstrated that a potential attacker has the ability to co-locate and detect co-location in public IaaS clouds. In 2011, Zhang et al. [36] 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. Recently, Zhang et al. [37] demonstrated that deduplication enables co-location detection and information extraction from co-located VMs in public PaaS clouds.

In follow-up to Ristenpart et al.'s work [29], Zhang et al. [33] and Varadarajan et al. [31] explored co-location detection in commercial public cloud in 2015. While the former study explores the cost of co-location in Amazon EC2, GCE and Microsoft Azure, the latter focuses only on Amazon EC2. In both studies, authors use the memory bus contention channel identified and explored by Wu et al. in 2012 [32] to detect co-location. In addition to the memory bus contention, [33] also counts the number of hops in traceroute trails of the VMs to achieve faster co-location detection and increase the probability of co-location.

Memory bus contention is created by executing an atomic operation that spans over multiple cache lines. Normally, when an atomic operation is performed, only the cache line that is operated on is locked. However, when the data spans more than one cache line, all memory operations on all cores and CPUs are flushed to ensure atomicity. Flushing of the memory operations slows the overall system and allows for a covert channel to be established between VMs. Capabilities of this channel however is limited only to slowing down co-located VM's operations. Therefore only useful *detecting* co-location, not for actually *exploiting*. Due to the limited nature of the memory bus contention channel, there have been no cryptographic key recovery attacks using this method. Moreover, when used only for co-location detection, results must be treaded lightly. Even though two VMs running on the same motherboard but on different sockets can be detected as co-located, the fact that they are running on different sockets immediately eliminates the ability to run any known cache side-channel attack. In other words, unless both the target and the attacker are running in the same CPU, the co-location is not exploitable, or at least not exploited for any known cryptographic attack unless VMs migrate from one socket to other.

Recovering cache slice selection methods: A basic technique based on hamming distances for recovering and exploiting **linear** cache slice selection was introduced in [18]. Irazoqui et al. [20] and Maurice et al. [27] 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. In this work, we use a more systematic approach by finding kernels in a vector space to recover a **non-linear** slice selection algorithm of order 4. In concurrent work, Yarom et al. [35] reverse engineered a 6 core Intel slice selection algorithm.

Side channel attacks: on RSA have been widely studied and explored as diverse side channels as time [25], [9], Branch Prediction Units [6], power [24], EM [12], [13], and even acoustic channels [14]. Attacks on RSA exploiting the cache side channel include [5], [34]. While [38], [11] do not explicitly target RSA but El Gamal instead, [11] targets a sliding window implementation and relies on LLC leakage, just like our attack. Unlike in that work, our work is performed in a public cloud, not in a controlled lab setup. We also present several new optimization techniques to reduce the number of needed observations to detect side channel leakage and noise reduction techniques. Recently, Gruss et al. [15] proved the ability of implementing last level cache template attacks in the presence of deduplication.

Due to the rich body of literature in side channel attacks there are also other works that have explored methods to reduce noise and recover keys from noisy CRT-RSA exponent leakage, such as [10] and [23]. However, note that cache based noise behaves significantly different from noise observed through an EM side channel (cf. [23]), and our techniques have been adapted to handle such cases well.

X. Ethical Concerns

Our experiments in public IaaS clouds were designed to conform with Amazon's acceptable use policy, the law, and proper ethic. In this work, we took all necessary precautions to make sure that we did not interfere with Amazon EC2's or its customer's services in any way. We did not, willingly or by accident obtain any information about Amazon EC2 customers.

During our co-location detection experiments, we only tried to achieve co-location with our own instances and not with other customers. We scheduled the co-location tests as well as the RSA key recovery to early in the morning when the usage is lowest to reduce the risk of interfering with other customers. And only after we detected and confirmed co-location, we have proceeded to the RSA key recovery step.

In RSA key recovery experiments, we synchronized our `Prime and Probe` side channel technique with the RSA execution, reducing the risk of obtaining any unauthorized

information. In addition to that, we ran our experiments with one hour intervals between 2 a.m. and 4 a.m. when other instances see minimum user activity. Hence, potential interference with other instances is minimized. We only monitored a single cache set out of more than 2,000 sets in one slice, causing minimal interference. Indeed, this is what made our attack viable. We believe that this load is much lower than regular Amazon EC2 workload.

The number of huge size pages allocated for experiments was limited to the minimum necessary to perform the attack. We allocated only 100 huge sized pages at most, reserving no more than 200 MB memory space. By doing so, we reduced any possible overhead that other co-located instances could have experienced. We performed our experiments so that we do not introduce any overhead other than the regular overhead that workload from regular instances. Furthermore, the slice selection algorithm was reverse engineered offline, on our lab setup, further limiting our involvement with other instances.

Finally, we have notified and shared our results with the AWS Security Team in June 2015.

XI. Conclusion

In conclusion, we show that even with advanced isolation techniques, resource sharing still poses a 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 become a practical attack vector for both co-location detection and data theft. Therefore, users have a responsibility to use latest improved software for their critical cryptographic operations. Additionally, placement policies for public cloud must be revised to diminish attacker's ability to co-locate with a targeted user. Even further, we believe that smarter cache management policies are needed both at the hardware and software levels to prevent side-channel leakages and future exploits.

XII. 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] Brumley, D., and Boneh, D. Remote Timing Attacks are Practical. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 1–14.

[10] Campagna, M. J., and Sethi, A. Key recovery method for crt implementation of rsa. Cryptology ePrint Archive, Report 2004/147. http://eprint.iacr.org/.

[11] 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*.

[12] 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.

[13] Genkin, D., Pachmanov, L., Pipman, I., and Tromer, E. Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation. Cryptology ePrint Archive, Report 2015/170, 2015. http://eprint.iacr.org/.

[14] Genkin, D., Shamir, A., and Tromer, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pp. 444–461.

[15] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 897–912.

[16] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. SP '11, pp. 490–505.

[17] Hamburg, M. Bit level error correction algorithm for rsa keys. Personal Communication, Cryptography Research, Inc., 2013.

[18] 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.

[19] 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)*.

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

[21] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies 1(1)*, 25–40.

[22] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.

[23] Jaffe, J., Kenworthy, G., and Hamburg, M. SPA/SEMA vulnerabilities of popular RSA-CRT sliding window implementations.

[24] 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.

[25] 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.

[26] Libgcrypt. The Libgcrypt reference manual. http://www.gnupg.org/documentation/manuals/gcrypt/.

[27] 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).

[28] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox - practical cache attacks in javascript. *CoRR abs/1502.07373* (2015).

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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/.

[36] 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*.

[37] 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*.

[38] 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*.

[39] ZHANG, Y., LI, M., BAI, K., YU, M., AND ZANG, W. Incentive compatible moving target defense against vm-colocation attacks in clouds. In *Information Security and Privacy Research*. 2012, pp. 388–399.